THE OFFICE OF ENTERPRISE TECHNOLOGY STRATEGIES

Statewide Technical Architecture

Implementation Guideline:

Enterprise Application Development Guidelines for the Microsoft^{Ò} .Net Framework

STATEWIDE TECHNICAL ARCHITECTURE

Implementation Guideline: Enterprise Application Development Guidelines for the Microsoft® .Net Framework

Revised Date:		Version:	1.0.0
Revision Approved Date:			
Date of Last Review:	March 6, 2003		
Date Retired:			
Architecture Interdependencies:			
Reviewer Notes: August 1, 2003			

Table of Contents

Introduction

The intent of this document is to outline the implementation guidelines that the State of North Carolina has adopted to ensure uniform and consistent implementations of Microsoft®.NET based solutions across the enterprise. Microsoft®.NET is a multifaceted and extremely flexible platform framework. Therefore, to ensure that application development is structured and consistent, development teams must adopt uniform policies and procedures that reflect enterprise implementation guidelines. To assist in this effort, there are tools provided within the platform that simplify the creation and adoption of these policy and procedures.

Detailed aspects of Microsoft®.NET development will not be covered in this document. There are resources listed in the following section that outline further support mechanisms for gaining greater depth and knowledge about the Microsoft®.NET framework. Also Microsoft has published prescriptive architecture guidance documents that are available on the Internet at http://msdn.microsoft.com/library that will cover the topics mentioned in this document in much greater detail. In fact, it should be noted that much of the material in this document has been derived from the PAG (Prescriptive Architecture Guidance) documents.

The key goal of this document is to outline implementation guidelines that when followed by the solution developers will lead to a well-designed Microsoft®.NET solution that has the flexibility to grow with changes in technology and can be maintained in an efficient and effective manner, which is a fundamental principle of the North Carolina Statewide Technical Architecture.

This document in no way indicates or implies a preference in a vendor or technology by the State. It is provided to ensure that agencies implementing Microsoft®.Net have the resources necessary to successfully deploy their systems in the most advantageous manner to both the agency and the State of North Carolina.

Support Mechanisms

The creation of enterprise class applications is complex and requires an in-depth knowledge and diverse skill set in areas such as application design, application programming, as well as a general understanding of networking and database technologies. To address this complexity, there are a number of support mechanisms that the State of North Carolina can leverage to create an enterprise application when using the Microsoft®.NET framework. To best assist the State of North Carolina in the utilization of these support mechanisms Microsoft has assigned two Account Executives that can bring together the right Microsoft resources and/or partners to help in different phases of the creation of a .NET enterprise application. Below is a list of the specific resources that are available and how and when they may be leveraged:

- **Microsoft Account Executive (State of North Carolina)** - Microsoft has two Account Executives for the State of North Carolina. Account Executives should be viewed as key strategic resources for Microsoft products and services. They focus on fostering a positive relationship between Microsoft and the State; and can leverage internal Microsoft resources as well as partners for guidance and support for Microsoft®.NET development efforts. As of this publication, the Microsoft Account Executives assigned to the State of North Carolina are:
    - **Jim Lorenz:** Phone: 919.474.4919;
      email jlorenz@microsoft.com
    - **Bill Durham:** Phone: 678-908-7544;
      email bdurham@microsoft.com

  Bill Durham is the primary contact. However, if your agency has already established a relationship with Jim Lorenz, he will remain as your primary contact.

- **Microsoft Developer Network (MSDN)** - MSDN is offered as both a web site and as a subscription.
    - The MSDN website (http://msdn.microsoft.com/) is one of the premier sites for information on the latest developments for web services and Microsoft®.NET. This document leverages material from the MSDN website.
    - The subscription essentially provides a developer with all the Microsoft tools and software necessary to craft a .NET enterprise application. Products include: Visual Studio.NET, SQL Server 2000, BizTalk Server 2002, etc. The products are delivered in CD or DVD media and are to be used for development purposes only – and may not be used for production systems. Licensed versions of the products must be purchased when it comes time to deploy. Contact one of the Microsoft Account Executives for more information.

- **MCS (Microsoft Consulting Services)** - MCS is the consulting side of Microsoft and is made up of highly skilled developers and architects that can help in the different phases of the application development process. Contact one of the Microsoft Account Executive for more information.

- **.NET Developer Quick Starts** – This Microsoft program offers customers all the required resources - Software, Technical Training, and consulting hours from MCS - to do a pilot on .NET. It's a great way to jump start development projects. Contact one of the Microsoft Account Executive for more information.

- **Two Day .NET Developer Training** - This two-day training can deliver on-site developer training at no cost. Course materials and trainers are provided. Contact one of the Microsoft Account Executive for more information.

- **Partners** - There are many partners that provide consulting services based on the Microsoft®.Net framework. Like MCS, these partners can provide the necessary skill set to help get customers up and running in all phases of development. Contact one of the Microsoft Account Executives for more information.

- **Product Support Services (PSS)** - For questions regarding specific programming and/or product issues, Microsoft PSS should be utilized. This is offered at various levels including subscription-based contracts that guarantee turnaround time for question and or issues. The knowledge base, provided by PSS (http://support.microsoft.com), can be used as an effective research tool.
- **Microsoft Certified Technical Education Centers (CTEC's)** - These are Microsoft partners who deliver technical training in the form of Microsoft Official Curriculum (MOC). Microsoft approved and designed courses guarantee a high quality of training. These courses are usually offered over a one-week period and will go into considerable technical depth. Courses are available for the Microsoft®.NET platform. For up to date information on the courses that are available and the centers that offer them, please visit: http://www.microsoft.com/traincert/training/.

- **Books and Magazines** - There are many books and magazines that are devoted to enterprise application development and/or Microsoft®.NET. MSDN Magazine is a great resource for finding how to articles and keeping up with the latest developments with Microsoft®.NET.

- **Local and Regional Architect Roundtables** - The goal of the Roundtable is to establish a dialogue with systems architects, designers and technology decision makers in the Carolinas. The Roundtable is made up of a small group of senior architects who are interested in having in-depth and confidential discussions on how to better architect business critical solutions using Microsoft technology. The membership of the roundtables represents influential companies in the community who are interested in improving the architecture of their business. Microsoft looks to this group to provide feedback on current Microsoft offerings, publications, and events and to provide direction on how to improve these offerings to better enable customers to meet the needs of their businesses. The local roundtables occur quarterly while regional events are annual.

- As of this publication a roundtable is being developed. Call your Microsoft Account Executive for more details.

Reference Architecture Design

The architecture of any enterprise application can be thought of in two different ways: Physical View and the Logical View. The Logical View addresses entities (components) that perform a certain function in the overall architecture - it is platform agnostic. The logical components that make up the Logical View map to the real world platform on which they live, gives rise to the Physical View. Both views will be presented for the Reference Architecture.

Physical View

Figure 1 illustrates the Physical View of the Reference .NET Architecture. It should be noted that this proposed architecture is offered as a best practice and will sometimes need to be deviated from the Statewide Technical Architecture (STA) as the application environmental factors demand, and with prior approval from ETS. The following annotated diagram is described below.

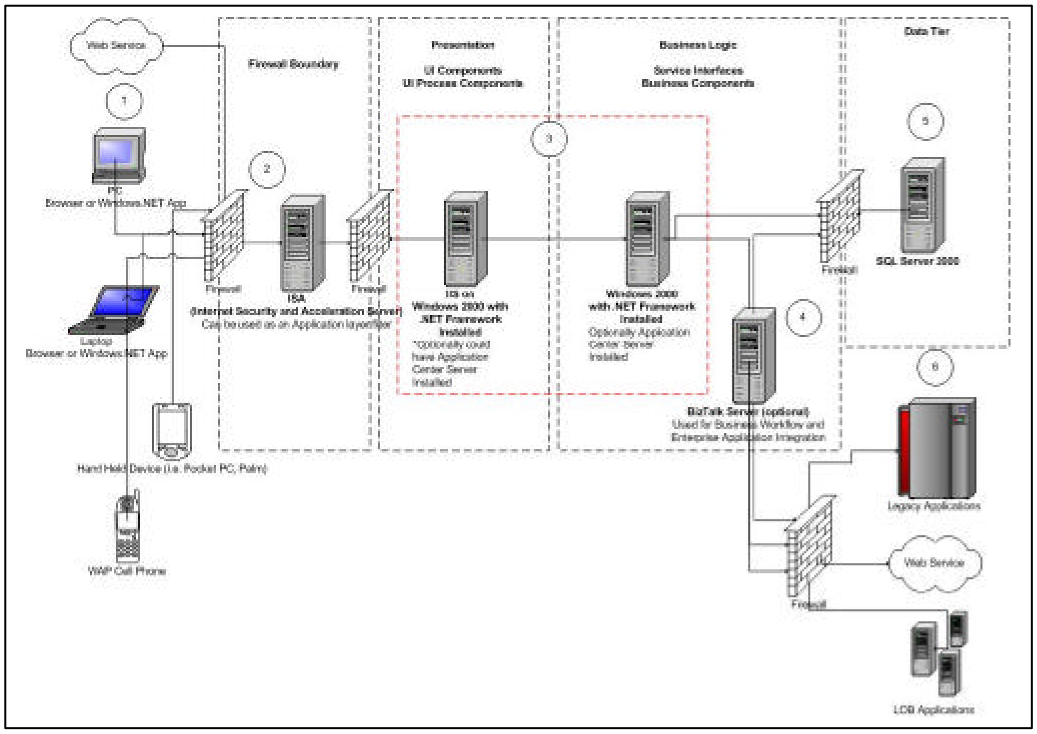

Web Service
Firewall Boundary
Presentation
UI Components
UI Process Components
Business Logic
Service Interfaces
Business Components
Data Tier
1
2
3
4
5
6
PC
Browser or Windows.NET App
Laptop
Browser or Windows.NET App
Hand Held Device (i.e. Pocket PC, Palm)
WAP Cell Phone
Firewall
ISA
(Internet Security and Acceleration Server)
Can be used as an Application layer filter
Firewall
IIS on
Windows 2000 with
.NET Framework
Installed
*Optionally could have Application Center Server Installed
Windows 2000
with .NET Framework
Installed
Optionally Application Center Server Installed
Firewall
SQL Server 2000
BizTalk Server (optional)
Used for Business Workflow and Enterprise Application Integration
Legacy Applications
Web Service
Firewall
LOB Applications

Figure 1 – Physical View of Proposed Architecture

1. **Clients:** The Architecture should be able to support the widest variety of clients. These include:
   - Standard PC browsers on laptops, desktops, workstations, and tablets.
   - Mobile Devices such as Pocket PC, Palm, WAP enabled phones, etc.
   - Web Services that need access to information provided by the application.

   The programming design points concerning the different types of clients will be addressed later in this document.
2. **Firewall Boundary:** For the edge firewall (FW), an existing FW may be used if already in place. In this scenario, the Internet Security and Acceleration (ISA) Server shown will provide application tier filtering and additional packet scanning capabilities. While this scenario has the ISA Server functioning as both an edge FW and application filter, the State's Technical Architecture requires a firewall independent of an ISA Server.
3. **Presentation and Business Tier:** The architecture shown here has these tiers on separate physical systems. This is a more secure approach than hosting both tiers on the same machine. The tradeoff is performance versus a more secure application. The programming design points concerning the different types of services provided in this tier will be addressed later in this document.
4. **Data Tier:** Microsoft SQL Server 2000 is the usual database of choice for Microsoft®.NET architecture. However any database can be used provided it has either a Managed Provider to the database (currently as of this publication, SQL 2000 and Oracle 9i provide this functionality; DB2/UDB will be available in early to mid 2003) or an OLE/DB provider for legacy access. The programming design points concerning the data tier will be addressed later in this document.

   Stored Procedures:

   To retain some level of database portability it is recommended that stored procedures only be used when there is a viable design goal that can only be solved by them. Otherwise, SQL statements for data access should be maintained in the business logic tier.

   Note:
   Remaining up-to-date on database service packs, patches, and hot fixes will help to ensure overall database security. Please review the IRMC policy – "Vulnerability Management Standard" for additional information.

   Database Security:

   Moving forward .NET applications must adhere to the state's Identity and Access Management Service (IAMS) authentication model for database security. IAMS will complement the existing database security using a role-based access control methodology and provide cross platform integration for users.
5. **External Applications:** Usually a distributed application will need to access legacy systems, LOB applications, and/or other web services. To achieve this, Microsoft provides BizTalk Server for the .NET platform or, as noted in item

4, an existing middleware/EAI solution may be leveraged. This is provided that it has a programmatic interface to provided messaging capabilities to and from the internal application and the external application.

.NET Security

The .NET Framework delivers a rich security infrastructure, which can be tiered on top of IAMS. IAMS is currently available to state agencies, see the "Authentication for the State of North Carolina" section of this document for more information. Complementary to IAMS, the .NET Framework provides developers a well-defined security model called role-based security in which all code runs on behalf of the invoking user/group. Permissions on these roles dictate whether or not they are able to execute or are explicitly unable to execute this code. Additionally, the .NET Framework provides security on this code (referred to code-access or evidence based security). Code-access security facilitates scenarios in which a user/group may be trusted to execute a component, but if this component is not trusted by the system, then access to the resource will be denied. This facility determines what permissions to grant to the executing code and checks that all code on the stack (in the caller chain) has the necessary permissions to execute the code. Additional details on .NET security and writing secure code in the .NET Framework can be found on http://msdn.microsoft.com/ and http://www.microsoft.com/security.

Note:
The term component is used in the sense of a piece or part of the overall solution. This includes compiled software components, such as Microsoft® .NET assemblies, and other software artifacts such as Web pages.

Logical View

In the following section, we will address the architectural design consideration of the Logical View. This section is targeted more toward developers and architects and is taken from the Architecture and Design of Distributed .NET Applications Prescriptive Architecture Guidance document available at:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp?frame=true

Component Types

An examination of most business solutions based on a tiered component model reveals several common component types. Figure 2 shows these component types in a comprehensive illustration that adheres to the state of North Carolina's existing technical architecture.

Although the list of component types shown in Figure 2 is not exhaustive, it represents the common types of software components found in most distributed solutions. These component types are described in depth throughout the remainder of this section.

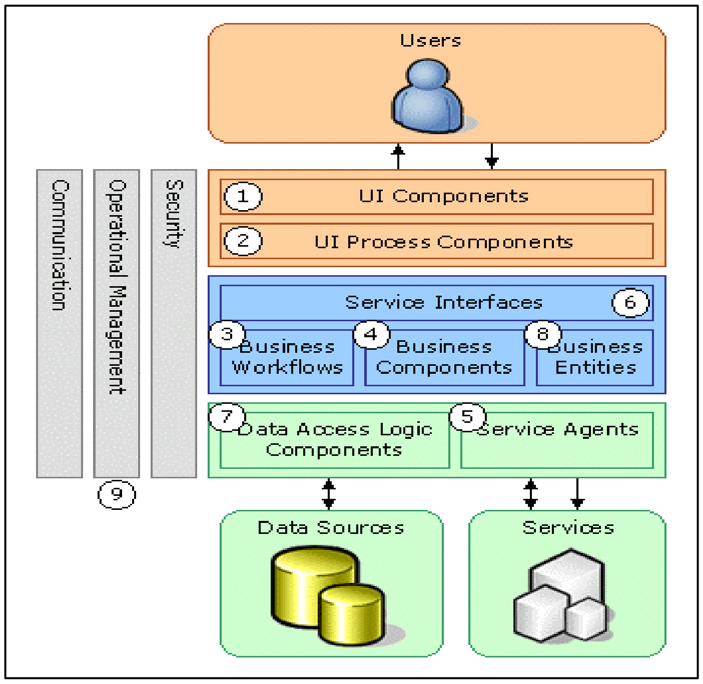


Figure 2 – Component types in the sample scenario

For the purpose of outlining the technologies involved in distributed architecture within the .NET platform, we have chosen a typical multi-tier Internet solution. In this solution, a user/citizen interacts through a user interface (UI), which in turn accesses business logic and data on behalf of the user/citizen. Also, in this design there are agency workers within the state domain that access the application via windows based UI's. The component types identified in the sample scenario design are:

1. **User interface (UI) components**. Most solutions need to provide a way for users to interact with the application. An example of a distributed application with multiple user interfaces is a Web site that lets customers view products and submit orders, and a Windows Forms-based application that lets agency representatives enter data. User interfaces are implemented using Windows Forms, Microsoft ASP.NET pages, controls, or any other technology used to display data for users and to acquire and validate incoming data.

   Proper UI design is an issue that should be addressed when developing an enterprise class application. There are many benefits for having a consistent look and feel across applications, especially within the same governmental entity. It is recommended that detailed and specific UI standards be defined for the following types of application design:

- Web Forms
- Windows Forms
- Mobile forms

Basic design goals of a good UI do not overload the user with too much information and utilize consistent placement of controls. The State of North Carolina is working with Microsoft to develop Enterprise Templates (standard routines and class modules) and the Visual Inheritance features of the Visual Studio.NET environment. Enterprise Templates offer the opportunity to wrap policy around the design/development environment, and encourages sharing the organization's knowledge, experiences, and assets at design time. The use of these templates removes the blank slate problems that plague the efficiency of many development projects. The Visual Inheritance features of Visual Studio.NET allow web and Windows Forms to inherit their look/feel from "base" forms. This inheritance allows changes to a "base" form to be propagated throughout the application.

2. **User process components**. In many cases, a user interaction with the system follows a predictable process. To help synchronize and orchestrate these user interactions, it is useful to drive the process using separate user process components. This prevents the process flow and state management logic from being hard-coded in the user interface elements themselves. The same basic user interaction "engine", therefore, can be reused by multiple user interfaces. For example, in a distributed application a procedure could be implemented for viewing product data that has the user select a category from a list of available product categories and then select an individual product in the chosen category to view its details. Similarly, when the user makes a purchase, the interaction follows a predictable process of gathering data from the user, in which the user first supplies details of the products to be purchased, then provides payment details, and then enters delivery details.

3. **Business workflows**. After a user process collects the required data, the data can be used to perform a business process. For example, after the product, payment, and delivery details are submitted to the distributed application, the process of taking payment and arranging delivery can begin. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. For example, the distributed application system would need to calculate the total value of the order, validate the credit card details, process the credit card payment, and arrange delivery of the goods. This process could take an indeterminate amount of time to complete, so the required tasks and the data required to perform them would have to be managed. Business workflows define and coordinate long-running, multi-step business processes, and they can be implemented using business process management tools that perform Orchestration functions.

4. **Service interfaces**. To expose business logic as a service, create service interfaces that support the communication contracts (e.g. message-based communication, formats, protocols, security, exceptions, etc.) that its different consumers require. For example, the credit card authorization service must expose a service interface that describes the functionality offered by the service and the required communication semantics for calling it. Service interfaces are sometimes referred to as *business facades*.

5. **Business components**. Regardless of whether a business process consists of a single step or an orchestrated workflow, the application will probably require components that implement business rules and perform business tasks. For example, in a distributed application, functionality would need to be implemented that calculates the total price of the goods ordered and adds the appropriate delivery charge. Business components implement the business logic of the application.

6. **Service agents**. When a business component needs to use functionality provided in an external service, code may need to be provided to manage the semantics of communicating with that particular service. For example, the business components of the distributed application described earlier could use a service agent to manage communication with the credit card authorization service, and use a second service agent to handle conversations with the courier service. Service agents isolate the idiosyncrasies of calling diverse services from the application, and can provide additional services, such as basic mapping between the format of the data exposed by the service and the format the application requires.

7. **Data access logic components**. Most applications and services will need to access a data store at some point during a business process. For example, the distributed application needs to retrieve product data from a database to display product details to the user, and it needs to insert order details into the database when a user places an order. It makes sense to abstract the logic necessary to access data in a separate tier of data access logic components. Doing so centralizes data access functionality and makes it easier to configure and maintain.

8. **Business entity components**: Most applications require data to be passed between components. For example, in the distributed application a list of products must be passed from the data access logic components to the user interface components so that the product list can be displayed to the users. The data is used to represent real-world business entities, such as products or orders. The business entities that are used internally in the application are usually data structures, such as DataSets, DataReaders, or eXtensible Markup Language (XML) streams, but they can also be implemented using custom object-oriented classes that represent the real-world entities the application has to work with, such as a product or an order.

9. **Components for security, operational management, and communication**: The application will probably also use components to perform exception management, to authorize users to perform certain tasks, and to communicate with other services and applications.

General Design Recommendations for Applications and Services

When designing an application or service, consider the following recommendations:

- Based on your agency's business needs, identify the components needed in your application from the list above. Some applications do not require certain components. For example, smaller applications that don't need to integrate with other services may not need business workflows or service agents. Similarly, applications that have only one user interface with a small number of elements may not require user process components.

- Design all components of a particular type to be as consistent as possible, using one design model or a small set of design models. This helps to preserve the predictability and maintainability of the design and implementation for all teams. In some cases, it may be hard to maintain a logical design due to technical environments (for example, if developing both ASP.NET- and Windows-based user interfaces); however, strive for consistency within each environment. In some cases, a base class for all components can be used that follows a similar pattern, such as data access logic components.

- Understand how components communicate with each other before choosing physical distribution boundaries. Keep coupling low and cohesion high by choosing remote communication components that are not chatty.

- Keep the format used for data exchange consistent within the application or service. If data representation formats are mixed, keep the number of formats low. For example, data may be returned in a DataReader from data access logic components to do fast rendering of data in Microsoft ASP.NET, but use DataSets for use in business processes. However, be aware that mixing XML strings, DataSets, serialized objects, DataReaders, and other formats in the same application will make the application more difficult to develop, extend, and maintain.

- Keep code that enforces policies (such as security, operational management, and communication restrictions) abstracted as much as possible from the application business logic. Try to rely on attributes, platform application programming interfaces (APIs), or utility components that provide "single line of code" access to functionality related to the policies, such as publishing exceptions, authorizing users, and so on.

- Determine the degree of tiering to enforce. In a strict tiering system, components in tier A cannot call components in tier C; they always call components in tier B. In a more relaxed tiering system, components in a tier can call components in other tiers that are not immediately below it. In all cases, try to avoid upstream calls and dependencies, in which tier C invokes tier B.

Designing Presentation Tiers

The presentation tier contains the components that are required to enable user interaction with the application. The most simple presentation tiers contain user interface components, such as ASP.NET Web Forms. For more complex user interactions, design user process components to orchestrate the user interface elements and control the user interaction. User process components are especially useful when the user interaction follows a predictable flow of steps, such as when a wizard is used to accomplish a task. Figure 3 shows the component types in the presentation tier.

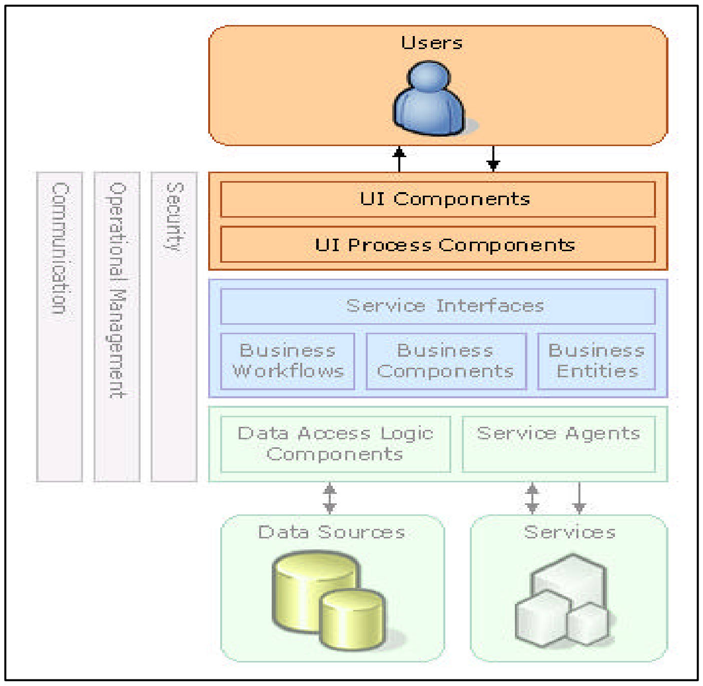


Figure 3 – Presentation Tier

Designing User Interface Components

User interfaces can be implemented in many ways. For example, our application requires a Web-based user interface and a Windows Forms-based user interface.

Other user interfaces include voice rendering, document-based programs, mobile client applications, etc. User interface components manage interaction with the user. They display data to the user, acquire data from the user, and interpret events that the user raises to act on business data, change the state of the user interface, or help the user progress in his task. The State of North Carolina's Standard Technical Architecture recommends implementing web-based user interfaces, rather than forms-based interfaces. This decreases client-side management while increasing platform selectivity.

User interfaces usually consist of a number of elements on a page or form that display data and accept user input. For example, a web-based application could contain a DataGrid control displaying a list of product categories, and a command button control used to indicate that the user wants to view the products in the selected category. When a user interacts with a user interface element, an event is raised that calls code in a controller function. The controller function, in turn, calls business components, data access logic components, or user process components to implement the desired action and retrieve any necessary data to be displayed. The controller function then updates the user interface elements appropriately. Figure 4 shows the design of a user interface.

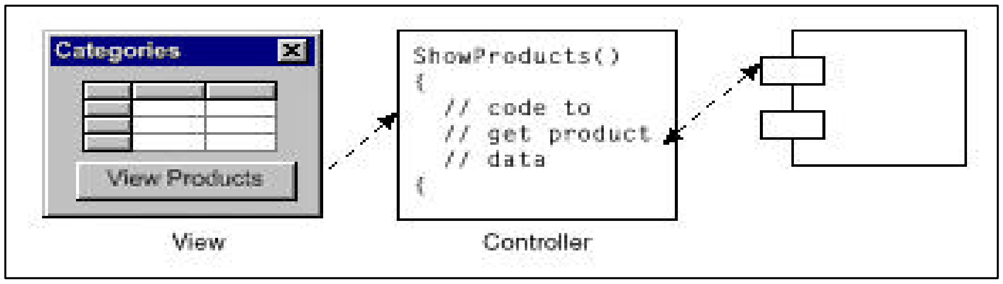


Figure 4 – User interface design

User Interface Component Functionality

User interface components must display data to users, acquire and validate data from user input, and interpret user gestures that indicate the user wants to perform an operation on the data. Additionally, the user interface should filter the available actions to let users perform only the operations that are appropriate at a certain point in time.

User interface components:

- Do not initiate, participate in, or vote on transactions.
- Have a reference to a current user process component if they need to display its data or act on its state.

When accepting user input, user interface components:

- Acquire data from users and assist in its entry by providing visual cues (such as tool tips), validation, and the appropriate controls for the task.
- Capture events from the user and call controller functions to tell the user interface components to change the way they display data, either by initiating an action on the current user process, or by changing the data of the current user process.
- Restrict the types of input a user can enter. For example, a **Quantity** field may limit user entries to numerical values.
- Perform data entry validation, for example by restricting the range of values that can be entered in a particular field, or by ensuring that mandatory data is entered.
- Perform simple mapping and convert information provided by the user controls to values needed by the underlying components to do their work (for example, a user interface component may display a product name but pass the product ID to underlying components).
- Interpret user gestures (such as a drag-and-drop operation or button clicks) and call a controller function.
- Cache the output of the user interface to avoid re-rendering it each time. In ASP.NET, specify caching on the output of a user interface component to avoid re-rendering it every time. If the application contains visual elements representing reference data that changes infrequently and is not used in transactional contexts, and these elements are shared across large numbers of users, cache them.

When rendering data, user interface components:

- Normally acquire and render data from business components. Very rarely will you render data from the data access logic components in the application.
- Perform formatting of values (such as formatting dates appropriately).
- Perform localization work on the rendered data (for example, using resource strings to display column headers in a grid in the appropriate language for the user's locale).
- Typically display data that pertains to a business entity. These entities are usually obtained from the user process component, but may also be obtained from the data components. UI components may render data by data-binding their display to the correct attributes and collections of the

entity components, if the entity is already available. Managing entity data as DataSets, is very simple to do. If custom business entity objects are implemented, it may be necessary to implement extra code to facilitate the data binding.

- Provide the user with connection status information. In the case of applications which use disconnected datasets (for example Mobile applications), the application needs to provide the user with status information, for example by indicating when an application is working in "disconnected" or "connected" mode. When the user connects to the network, the application notifies the user that data has been synchronized and whether there were any collisions.

- Customize the appearance of the application based on user preferences or the kind of client device used.

- Provide “undo” functionality. Many applications need to let a user undo certain operations. This is usually performed by keeping a fixed-length stack of "old value-new value" data for specific data items or entities. When the operation involves a financial process, do not expose the compensation as a simple undo function, but as an explicit operation.

- Provide clipboard functionality. In many Windows-based applications, it is useful to provide clipboard capabilities for more than just scalar values—for example, it may be necessary to let users copy and paste a full customer object. Such functionality is usually implemented by placing XML strings in the Clipboard in Windows, or by having a global object that keeps the data in memory if the clipboard is application-specific.

- Show long lists of data as paged sets. It is common, particularly in Web applications, to show long lists of data as paged sets. Use a helper component that will keep track of the current page the user is on and thus invoke the data access logic component paged query function with the appropriate values for page size and current page. Paging can occur without interaction of the user process component.

Windows Desktop User Interfaces

Windows user interfaces are used to provide disconnected or offline capabilities or rich user interaction, or even integration with the user interfaces of other applications. Windows user interfaces can take advantage of a wide range of state management and persistence options and can access local processing power. There are three main families of standalone user interfaces: "full-blown" Windows-based applications, Windows-based applications that include embedded HTML, and application plug-ins that can be used within a host application's user interface:

- **"Full-blown" desktop / tablet PC user interfaces built with Windows Forms**

  Building a Windows-based application often involves building an application with Windows Forms and controls where the application provides all or most of the data rendering functionality. This provides a great deal of control over the user experience and total control over the look and feel of the application. However, it is tied to a client platform, and the application must be deployed to the users (even if the application is deployed by downloading it over an HTTP connection.). Also, if there is any business logic or functionality within the Windows-based application, they must be deployed the same way. This must be considered when a large number of users will be accessing the application. For this and other reasons related to deployment and maintenance, web-based applications are typically preferred by the state.

- **Embedded HTML**

  The entire user interface may be implemented using Windows Forms, or additional embedded HTML in the Windows-based applications may be selected. Embedded HTML allows for greater run-time flexibility (because the HTML may be loaded from external resources or even a database in connected scenarios) and user customization. However, carefully consideration must be exercised regarding how to prevent malicious script from being introduced in the HTML. In addition, coding is required to load the HTML, display it, and hook up the events from the control with application functions.

- **Application plug-ins**

  Use cases may suggest that the user interface of an application could be better implemented as a plug-in for other applications, such as Microsoft Office, AutoCAD, Customer Relationship Management (CRM) solutions, engineering tools, etc. In this case, leverage all of the data acquisition and display logic of the host application and provide only the code to gather the data and work with the business logic.

  Most modern applications support plug-ins as either Component Object Model (COM) or .NET objects supporting a specified interface, or as embedded development environments (such as the Microsoft Visual Basic® development system, which is widely supported in most common Windows-based applications) that can, in turn, invoke custom objects. Some embedded environments (including Visual Basic) even provide a forms engine that enables an extension to the user interface experience beyond that provided by the host application. For more information about using Visual Basic in host applications, see "Microsoft Visual Basic for Applications and Windows DNA 2000" on MSDN (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndna/html/vba4dna.asp).

For information about working with .NET from Microsoft Office, see "Microsoft Office and .NET Interoperability" on MSDN

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnofftalk/html/office11012001.asp).

When creating a Windows Forms-based application, consider the following recommendations:

- Rely on data binding to keep data synchronized across multiple forms that are open simultaneously. This alleviates the need to write complex data synchronization code.

- Try to avoid hard-coding relationships between forms, and rely on the user process component to open them and synchronize data and events. Special care should be given to avoid hard-coding relationships from child forms to parent forms. For example, a product details window can be reused from other places in the application, not just from an order entry form; avoid implementing functionality in the product details form that links directly to the order entry form. This makes the user interface elements more reusable.

- Implement error handlers in forms. Doing so prevents the user from seeing an unfriendly .NET exception window and having the application fail if exceptions were not handled elsewhere. All event handlers and controller functions should include exception catches. Additionally, it may be desirable to create a custom exception class for the user interface that includes metadata to indicate whether the failed operation can be retried or canceled.

- Validate user input in the user interface. Validation should occur at the stages in the user's task or process that allow point-in-time validations (allowing the user to enter some of the required data, continue with a separate task, and return to the current task). In some cases, proactively enable and disable controls and visually cue the user when invalid data is entered. Validating user input in the user interface prevents unnecessary round trips to server-side components when invalid data has been entered.

- When creating custom user controls, expose only the public properties and methods that are actually needed. This makes the components more maintainable.

- Implement controller functions as separate functions in Windows Forms or in .NET classes that will be deployed with the client. Do not implement controller functionality directly in control event handlers. Writing controller logic in event handlers reduces the maintainability of the application, because it may be required to invoke the same function from other events in the future.

Mobile Device User Interfaces

Mobile devices such as handheld PCs, Wireless Application Protocol (WAP) phones, and iMode devices are becoming increasingly popular, and building user interfaces for a mobile form factor presents its own unique challenges. Therefore a strong business case for the development, implementation, and support for this type of interface should be justified before deciding to utilize this UI option.

In general, a user interface for a mobile device must be able to display information on a much smaller screen than other common applications, and it must offer acceptable usability for the devices being targeted. Because user interaction can be awkward on many mobile devices, particularly mobile phones, design mobile user interfaces with minimal data input requirements. A common strategy is to combine the use of mobile devices with a full-sized Web- or Windows-based application and allow users to pre-register data through the desktop-based client, and then select it when using the mobile client. For example, an e-commerce application may allow users to register credit card details through the Web site, so that a pre-registered credit card can be selected from a list when orders are placed from a mobile device (thus avoiding the requirement to enter full credit card details using a mobile telephone keypad or personal digital assistant [PDA] stylus).

Web User Interfaces

A wide range of mobile devices support Internet browsing. Some use micro browsers that support a subset of HTML 3.2, some require data to be sent in Wireless Markup Language (WML), and some support other standards such as Compact HTML (cHTML). The Microsoft Mobile Internet Toolkit can be used to create ASP.NET-based Web applications that send the appropriate markup standard to each client based on the device type as identified in the request header. Utilization of the toolkit allows for the creation of a single Web application that targets a multitude of different mobile clients including Pocket PC, WAP phones, iMode phones, and others.

As with other kinds of user interface, try to minimize the possibility of users entering invalid data in a mobile Web page. The Mobile Internet Toolkit includes client-side validation controls such as the CompareValidator, CustomValidator, RegularExpressionValidator, and RequiredFieldValidator controls, which can be used with multiple client device types. The properties of input fields can also be used such as Textbox controls to limit the kind of input accepted (for example by accepting only numeric input). However, always allow for client devices that may not support client-side validation, and perform additional checks after the data has been posted to the server.

For more information about the Mobile Internet Toolkit, see the Microsoft Mobile Internet Toolkit page on MSDN

(http://msdn.microsoft.com/vstudio/device/mitdefault.asp).

Smart Device User Interfaces

The Pocket PC is a feature-rich device based on the Windows CE operating system on which both disconnected and connected user interfaces can be developed. In most cases Pocket PCs utilize wireless connections to communicate with the application). The Pocket PC platform includes handheld PDA devices and smart phones, which combine PDA and phone features.

Microsoft provides the .NET Compact Framework for Pocket PC and other Windows CE platforms. The compact framework contains a subset of the full .NET Framework and allows the development of rich .NET–based applications for mobile devices. Developers can use the Smart Device Extensions for Visual Studio .NET to create applications that target the .NET Compact Framework.

As with regular Windows-based user interfaces, provide exception handling in the mobile device to inform the user when an operation fails and allow the user to retry or cancel it as appropriate.

No input validation controls are provided in the Smart Device Extensions for Microsoft Visual Studio® .NET; therefore, implement client-side validation logic to ensure that all data entry is valid.

For more resources for Pocket PC platform development and the .NET Compact Framework, see the Smart Device Extensions page on MSDN

(http://msdn.microsoft.com/vstudio/device/smartdev.asp).

Another mobile form factor for rich clients that may be considered is the Tablet PC. Tablet PCs are Windows XP–based portable devices that support user interaction through a "pen and ink" metaphor in which the user "draws" and "writes" on the screen. Since the Tablet PC is based on Windows XP, the full .NET Framework can be leveraged. An additional API for handling "pen and ink" interactions is also available. For more information about designing applications for the Tablet PC, see Design Recommendations for Exploiting the Pocket PC on MSDN

(http://msdn.microsoft.com/library/en-us/tpcsdk10/html/whitepapers/designguide/tbconuxdgformfactorpenandink.asp).

Document-based User Interfaces

Rather than build a custom Windows-based desktop application to facilitate user interaction, it may make more sense in some circumstances to allow users to interact with the system through documents created in common productivity tools such as Microsoft Word or Microsoft Excel. Documents are a common metaphor for working with data. In some applications, it may be beneficial to have users enter or view data in document form in the tools they commonly use. Consider the following document-based solutions:

- **Reporting data**. An application (Windows- or Web-based) may provide the user with a feature that lets him or her see data in a document of the appropriate type—for example, showing invoice data as a Word document, or a price list as an Excel spreadsheet.
- **Gathering data**. Agency representatives may enter information for citizens who telephone via Excel spreadsheets to create a customer order document, and then submit the document to a business process.

There are two common ways to integrate a document experience in an application, each broken down into two common scenarios: gathering data from users and reporting data to users.

Working with Documents from the Outside

Documents "from the outside" may be treated as an entity. In this scenario, code operates on a document that has no specific awareness of the application. This approach has the advantage that the document file may be preserved beyond a specific session. This model is with "freeform" areas in the document that the application doesn't need to deal with but may need to preserve. For example, choosing this model allows users to enter information in a document on a mobile device and take advantage of the Pocket PC ActiveSync capabilities to synchronize data between the document on the mobile device and a document kept on the server. In this design model, user interfaces perform the following functions:

- **Gathering data.** A user can enter information in a document, starting with a blank document, or most typically, starting with a predefined template that has specific fields.

  The user then submits the document to a Windows-based application or uploads it to a Web-based application. The application scans the document's data and fields through the document's object model, and then performs the necessary actions.

  At this point, it may be decided to either preserve the document after processing or to dispose of it. Typically, documents are preserved to maintain a tracking history or to save additional data that the user has entered in freeform areas.

- **Reporting data.** In this case, a Windows- or Web-based user interface provides a way to generate a document that shows some data, such as a report. The reporting code will usually take data from the ongoing user process, business process, and/or data access logic components and either call macros on the document application to inject the data and format it, or save a document with the correct file format and then return it to the user. The document may be returned by saving it to disk and providing a link to it (the document must be saved in a central store in load-balanced Web farms), or by including it as part of the response.

When returning documents in Web-based applications, decide whether to display the document in the browser for the user to view, or to present the user with an option to save the document to disk. This is usually controlled by setting the correct MIME type on the response of an ASP.NET page. In Web environments, follow file-naming conventions carefully to prevent concurrent users from overwriting each other's files.

Working with Documents from the Inside

When providing an integrated user experience within the document, embed the application logic in the document itself. In this design model, the user interface performs the following functions:

- **Gathering data.** Users can enter data in documents with predefined forms, and then specific macros can be invoked on the template to gather the right data and invoke business or user process components. This approach provides a more integrated user experience, because the user can just click a custom button or menu option in the host application to perform the work, rather than having to submit the entire document.

- **Reporting data.** Custom menu entries and buttons in documents that gather some data from the server and display it may be implemented. Smart tags may also be used in documents to provide rich inline integration functionality across all Microsoft Office productivity tools. For example, a smart tag that lets users display full customer contact information from the CRM database whenever an agency representative types in a citizen name in the document.

Regardless of whether working with a document from the inside or from the outside, provide validation logic to ensure that all user input is valid. This is achieved in part by limiting the data types of fields, but in most cases custom functionality must be implemented to check user input, and display error messages when invalid data is detected. Microsoft Office–based documents can include custom macros to provide this functionality.

For information about how to integrate a purely Office-based UI with business processes, see "Microsoft Office XP Resource Kit for BizTalk Server Version 2.0"

(http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/743/msdncompositedoc.xml).

For more information about working with Office and .NET, see MSDN. The following two articles will help with getting started with Office and .NET-based application development:

- "Introducing .NET to Office Developers" (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnofftalk/html/office10042001.asp)
- "Microsoft Office and .NET Interoperability" (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnofftalk/html/office11012001.asp)

Document-based workflows can be managed by taking advantage of the services provided by Microsoft SharePoint Portal™. This product can manage the user process and provides rich metadata and search capabilities.

Note:
Implementing a user interaction with user process components is not a trivial task. Before committing to this approach, carefully evaluate whether or not the application requires the level of orchestration and abstraction provided by user process components.

Designing User Process Components

A user interaction with an application may follow a predictable process; for example, the application may require users to enter product details, view the total price, enter payment details, and finally enter delivery address information. This process involves displaying and accepting input from a number of user interface elements, and the state for the process (which products have been ordered, the credit card details, and so on) must be maintained between each transition from one step in the process to another. To help coordinate the user process and handle the state management required when displaying multiple user interface pages or forms, create user process components.

User process components are typically implemented as .NET classes that expose methods that can be called by user interfaces. Each method encapsulates the logic necessary to perform a specific action in the user process. The user interface creates an instance of the user process component and uses it to transition through the steps of the process. The names of the particular forms or ASP.NET pages to be displayed for each step in the process can be hard-coded in the user process component (thus tightly binding it to specific user interface implementations), preferably they can be retrieved from a metadata store such as a configuration file (making it easier to reuse the user process component from multiple user interface implementations). Designing user process components to be used from multiple user interfaces will result in a more complex implementation in order to isolate device-specific issues, but can help distribute the user interface development work between multiple teams, each using the same user process component.

User process components coordinate the display of user interface elements. They are abstracted from the data rendering and acquisition functionality provided in the user interface components. Design them with globalization in mind, to allow for localization to be implemented in the user interface. For example, endeavor to use culture-neutral data formats and use Unicode string formats internally to make it easier to consume the user process components from a localized user interface.

Separating the user interaction functionality into user interface and user process components provides the following advantages:

- Long-running user interaction state is more easily persisted, allowing a user interaction to be abandoned and resumed, possibly even using a different user interface. For example, a customer could add some items to a shopping cart using the Web-based user interface, and then call an agency representative later to complete the order.

- The same user process can be reused by multiple user interfaces. For example, in the application, the same user process could be used to add a product to a shopping basket from both the Web-based user interface and the Windows Forms-based application.

An unstructured approach to designing user interface logic may result in undesirable situations as the size of the application grows or new requirements are introduced. If a specific user interface must be added for a given device, the data flow and control logic may need to be redesigned.

Partitioning the user interaction flow from the activities of rendering data and gathering data from the user can increase the application's maintainability and provide a clean design to which seemingly complex features can easily be added such as support for offline work. Figure 1.4 shows how the user interface and user process can be abstracted from one another.

Another good reason for properly separating/designing user interaction is for security and performance reasons. For example, user interfaces should be design to only require SSL when necessary such as for a shopping cart or HIPPA related information.

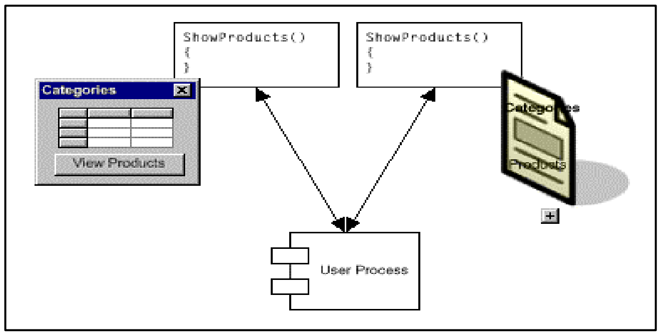


Figure 5 – User interfaces and user process components

User process components help to resolve the following user interface design issues:

- **Handling concurrent user activities**. Some applications may allow users to perform multiple tasks at the same time by making more than one user interface element available. For example, a Windows-based application may display multiple forms, or a Web application may open a second browser window.

  User process components simplify the state management of multiple ongoing processes by encapsulating all the state needed for the process in a single component. Each user interface can be mapped element to a particular instance of the user process by incorporating a custom process identifier into the design.

- **Using multiple panes for one activity**. If multiple windows or panes are used in a particular user activity, it is important to keep them synchronized. In a Web application, a user interface usually displays a set of elements in a same page (which may include frames) for a given user activity. However, in rich client applications, several non-modal windows affecting just one particular process may be necessary. For example, a product category selector window floating in the application that to specify a particular category, the products in which will be displayed in another window.

  User process components help implement this kind of user interface by centralizing the state for all windows in a single location. Synchronization can be further simplified across multiple user interface elements by using data bindable formats for state data.

- **Isolating long-running user activities from business-related state**. Some user processes can be paused and resumed later. The intermediate state of the user process should generally be stored separately from the application's business data. For example, a user could specify some of the information required to place an order, and then resume the checkout process at a later time. The pending order data should be persisted separately from the data relating to completed orders, allowing business operations to be performed on the completed order data (for example, counting the number of orders placed in the current month) without having to implement complex filtering rules to avoid operating on incomplete orders.

  User activities, just like business processes, may have a "timeout" specified, when the activity has to be cancelled and the right compensatory actions should be taken on the business process.

  Design user process components with the ability to be serialized, or to store their state separately from the application's business data.

Separating a User Process from the User Interface

To separate a user process from the user interface:

1. Identify the business process or processes that the user interface process will help to accomplish. Identify how the user sees this as a task (this can usually be done by consulting the sequence diagrams that are created as part of the requirements analysis).

2. Identify the data needed by the business processes. The user process will need to be able to submit this data when necessary.

3. Identify additional states needed to maintain throughout the user activity to assist rendering and data capture in the user interface.

4. Design the visual flow of the user process and the way that each user interface element receives or gives control flow.

Implement code to map a particular user interface session to the related user process:

- ASP.NET pages will have to obtain the current user process by getting a reference from the Session object, or by re-hydrating the process from another storage medium, such as a database. This reference is needed in event handlers for the controls on the Web page.

- Windows or controls need to keep a reference to the current user process component. Keep this reference in a member variable. Do not keep it in a global variable; composing user interfaces will become very complicated as the application user interface grows.

User Process Component Functionality

User process components:

- Provide a simple way to combine user interface elements into user interaction flows without requiring redevelopment of data flow and control logic.
- Separate the conceptual user interaction flow from the implementation or device where it occurs.
- Encapsulate how exceptions may affect the user process flow.
- Keep track of the current state of the user interaction.

- Should not start or participate in transactions. They keep internal data related to application business logic and their internal state, persisting the data as required.
- Maintain internal business-related state, usually holding on to one or more business entities that are affected by the user interaction. Keep multiple entities in private variables or in an internal array or appropriate collection type. In the case of an ASP.NET-based application, references to this data may be kept in the Session object. However, doing so limits the useful lifetime of the user process.
- May provide a "save and continue later" feature by which a particular user interaction may be restarted in another session. Implement this functionality by saving the internal state of the user process component in some persistent form and providing the user with a way to continue a particular activity later. Create a custom task manager utility component to control the current activation state of the process. The user process state can be stored in one of a number of places:
    - If the user process can be continued from other devices or computers, store it centrally in a location such as a database.
    - If running in a disconnected environment, the user process state will need to be stored locally on the user device.
    - If the user interface process is running in a Web farm, store any required state on a central server location, so that it can be continued from any server in the farm.
- May initialize internal state by calling a business process component or data access logic components.
- Typically will not be implemented as enterprise services components. The only reason to do so would be to use the enterprise services role-based authorization capabilities.
- Can be started by a custom utility component that manages the menus in the application.

User Process Component Interface Design

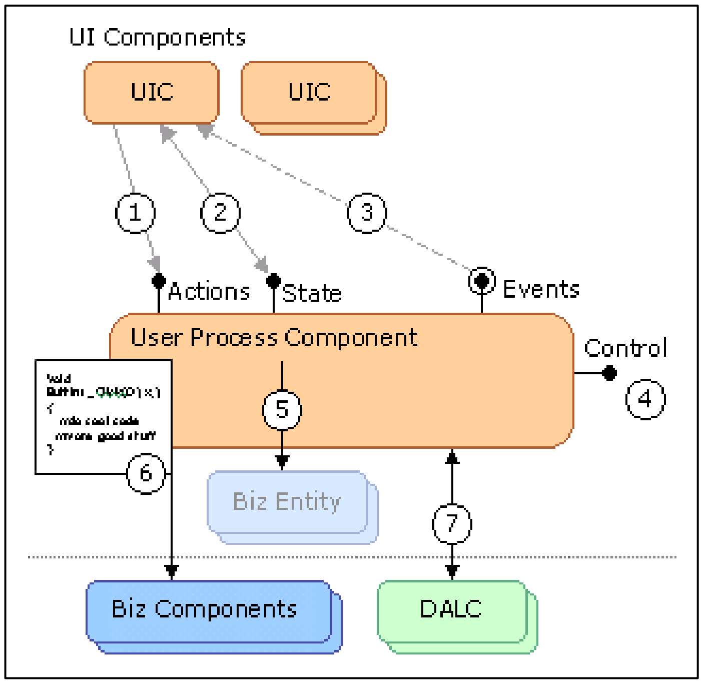
The interface of the user process components can expose the following types of functionality, as shown in Figure 6.

- **User process "actions" (1).** These are the interface of actions that typically trigger a change in the state of the user process. Actions are implemented in user process component methods, as demonstrated by the ShowOrder, EnterPaymentDetails, PlaceOrder, and Finish methods in the code sample discussed earlier. Encapsulate calls to business components in these action methods where possible (6).

- **State access methods (2).** Access the business-specific and business-agnostic state of the user process by using fine-grained get and set properties that expose one value, or by exposing the set of business entities that the user process deals with (5). For example, in the code sample discussed earlier, the user process state can be retrieved through public DataSet properties.

- **State change events (3).** These events are fired whenever the business-related state or business-agnostic state of the user process changes. It may be necessary to implement these change notifications manually. In other cases, data may be stored through a mechanism that already does this intrinsically (for example, a DataSet fires events whenever its data changes).

- **Control functions that allow start, pause, restart, and cancel a particular user processes (4).** These functions should be kept separate, but can be intermixed with the user process actions. For example, the code sample discussed earlier contains SaveToDataBase and ResumeCheckout methods. Control methods could load required reference data for the UI (such as the information needed to fill a combo box) from data access logic components (7) or delegate this work to the user interface component (forms, controls, ASP.NET pages) that needs the data.



```
void
Button_OnClick(x)
{
  //do cool code
  //more good stuff
}
```

Figure 6 – User process component design

General Recommendations for User Process Components

When designing user process components, consider the following recommendations:

- Decide whether to manage user processes as components that are separate from the user interface implementation. Separate user processes are most needed in applications with a high number of user interface dialog boxes, or in applications in which the user processes may be subject to customization and may benefit from a plug-in approach.
- Choose where to store the state of the user process:
    - If the process is running in a connected fashion, store interim state for long-running processes in a central SQL Server database; in disconnected scenarios, store it in local XML files, isolated storage, or local Microsoft SQL Server™ 2000 Desktop Engine (MSDE) databases. On Pocket PC devices, store state in a SQL Server CE database. Confidential information needs to be properly secure both during transmission and while being stored on the client. Encryption, Secure Sockets Layer and proper database security (no guest users and blank passwords) can achieve this.
    - If the process is not long running and does not need to be recovered in case of a problem, persist the state in memory. For user interfaces built for rich clients, keep the state in memory. For Web applications, store the user process state in the Session object of ASP.NET. If In a Web farm, store the session in a central state server or a SQL Server database. ASP.NET will clean up SQL Server-stored session to prevent the buildup of stale data.
- Design user process components with the ability to be serialized. This will help implement any persistence scheme.
- Include exception handling in user process components, and propagate exceptions to the user interface. Exceptions that are thrown by the user process components should be caught by user interface components and published as described in the "Security, Operational Management, and Communications Policies."

Network Connectivity and Offline Applications

In many cases, an application will require support for offline operations when network connectivity is unavailable. For example, many mobile applications, including rich clients for Pocket PC or Table PC devices, must be able to function when the user is disconnected from the corporate network. Offline applications must rely on local data and user process state to perform their work. When designing offline applications, follow the general guidelines in the following discussion.

The online and offline status should be displayed to the user. This is usually done in status bars or title bars or with visual cues around user interface elements that require a connection to the server.

The development of most of the application user interface should be reusable, with little or no modification needed to support offline scenarios. While offline, the application will not have:

- Access to online data returned by data access logic components.
- The ability to invoke business processes synchronously. As a result, the application will not know whether the call succeeded or be able to use any returned data.

If the application does not implement a fully message-based interface to the servers but relies on synchronously acquiring data and knowing the results of business processes (as most of today's applications do), do the following to provide the illusion of connectivity:

- Implement a local cache for read-only reference data that relates to the user's activities. Then implement an offline data access logic component that implements exactly the same queries as the server-side data access logic components but accesses the local storage. Implement the local cache as a desktop MSDE database. This enables the reuse of the design and implementation of the main SQL Server schemas and stored procedures. However, MSDE affects the global state of the computer it is installed on, there may be trouble accessing it from applications configured for semi-trust. In many scenarios, using MSDE may be overkill for state persistence requirements, and storing data in an XML file or persisted dataset may be a better solution.
- Implement an offline business component that has the same interface as business components, but takes the submitted data and places it in a store-and-forward, reliable messaging system such as Message Queuing. This offline component may then return nothing or a preset value to its caller.
- Implement UI functionality that provides a way to inspect the business action "outbox" and possibly delete messages in it. If Message Queuing is

used to queue offline messages, set the correct permissions on the queue to do this from the application.

- Design the application's transactions to accommodate message-based UI interactions. Take extra care to manage optimistic locking and the results of transactions based on stale data. A common technique for performing updates is to submit both the old and new data, and to let the related business process or data access logic component eventually resolve any conflicts. For business processes, the submission may include critical reference data that the business logic uses to decide whether or not to let the data through. For example, product prices can be included alongside product IDs and quantities when submitting an order. For a more detailed discussion of optimistic locking, see "Designing Data Tier Components and Passing Data Through Tiers" on MSDN

  (http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/BOAGag.asp?frame=true).

- Let the user persist the state of the application's user processes to disk and resume them later.

The advent of mobile devices based on IP networking, wireless security standard evolution, the 801.11 standard, IPv6, the Tablet PC, and other technologies will make wireless networks more popular. The issue with wireless networks is that with today's technology, they cannot guarantee connectivity with high confidence in all areas. For example, building structure, nearby machinery, and other factors may cause permanent and transient "dark zones" in the network. If designing an application for use in a wireless environment, consider designing it as a message-based, offline application, to prevent an experience full of exceptions and retries. For example, an application could be designed so that an offline user can enter data through the same user interface as when connected, and the data can be stored in a local database or queued and synchronized later, when the user reconnects. SQL Server supports replication, which can be used to automate the synchronization of data in a loosely coupled fashion, allowing data to be downloaded to the offline device while connected, modified while disconnected, and resynchronized when reconnected. Microsoft Message Queuing allows data to be encapsulated in a message and queued on the disconnected device for submission to a server-side queue when connected. Components of the server will then read the message from the queue and process it. Using local queues or SQL Server replication to handle communication of user input to the server can help mitigate connectivity issues, even when the application is nominally connected. Where a more tightly coupled approach is required, use transactions and custom logging to ensure data integrity.

When data synchronization occurs between a disconnected (or loosely coupled) application and a server, the following are security considerations:

- Message Queuing provides its own authorization model, based on Windows authentication. If the application relies on custom, application-managed authentication, client-side components will need to sign the documents that are submitted to the server.
- The client cannot be impersonated on the server if data is submitted through a queue.
- If SQL Server replication is used, specify an account with permission to access the SQL Server databases on the server. When replicating from SQL Server CE on a mobile device, a secure connection to the Internet Information Services (IIS) site containing the SQL Server CE Server Agent must be established. For more information about configuring SQL Server replication, see the documentation supplied with SQL Server and SQL Server CE.
- If network communication takes place over an HTTP connection, use Secure Sockets Layer (SSL) to secure the channel.

Notification to Users and Business Process-to-User Communication

The application may be required to notify users about specific events. As the communication capabilities of the Internet grow, more options for notifying users will be available. Common technologies currently include e-mail, instant messaging, cell phone messaging, paging, etc.

Instant notification may involve many possible notification technologies and the use of presence services to detect the appropriate way to contact a user. Microsoft Patterns & Practices has released a reference architecture that covers this scenario. It is available on MSDN at

http://msdn.microsoft.com/library/en-us/dnenra/html/enraelp.asp.

Designing Business Tiers

The core of the application is the business functionality it provides. An application performs a business process that consists of one or more tasks. In the simplest cases, each task can be encapsulated in a method of a .NET component, and called synchronously or asynchronously. For more complex business processes that require multiple steps and long running transactions, the application needs to have some way of orchestrating the business tasks and storing state until the process has completed. In these scenarios, BizTalk Server Orchestration and other Enterprise Application Integrations products that support BPEL4WS (Business Processing Execution Language for Web Services) can define the workflow for the business process. The BizTalk Server schedule that implements the workflow can then use BizTalk Server messaging functionality or call .NET business components to perform each task, as it is required.

Developers should design the logic in business tiers to be used directly by presentation components or to be encapsulated as a service and called through a service interface, which coordinates the asynchronous conversation with the service's callers and invokes the BizTalk Server workflow or business components. The core of the business logic is sometimes also referred to as *domain* logic. Business components may also make requests of external services, in which case it may be necessary to implement service agents to manage the conversation required for the particular business task performed by each service. Figure 7 shows the business tiers of an application.

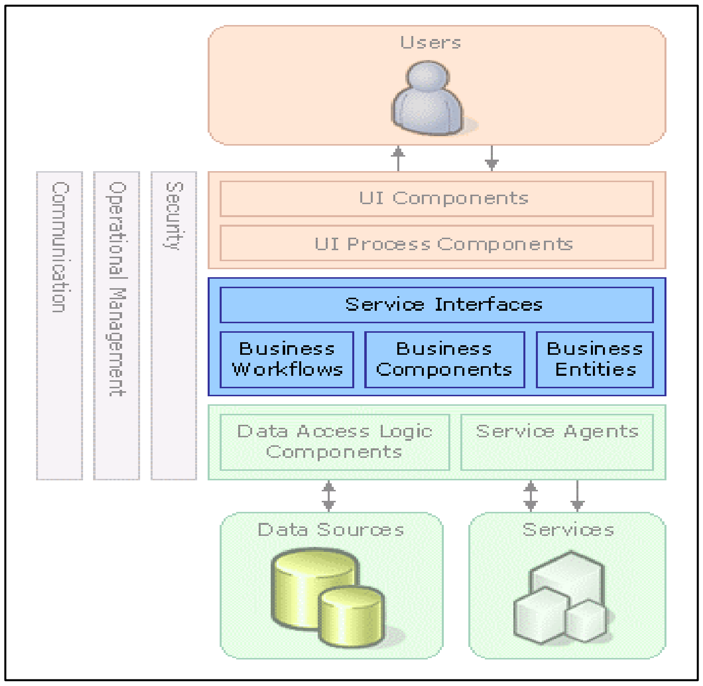


Figure 7 – Business component tiers

Business Components and Workflows

When implementing business functionality, decide if orchestration of the business process is necessary or if a set of business components will be sufficient.

Business workflows should be used (implemented with BizTalk Orchestration) to:

- Manage a process that involves multiple steps and long-running transactions.

- Expose an interface that implements a business process enabling the application to engage in a conversation or contract with other services.
- Take advantage of the broad range of adaptors and connectors for multiple technologies that are available for BizTalk Server.

Implement the business process using only business components when:

- It is not necessary to maintain conversation state beyond the business activity, and the business functionality can be implemented as a single atomic transaction.
- It is necessary to encapsulate functionality and logic that can be reused from many business processes.
- The business logic that must be implemented is computationally intensive or requires fine-grained control of data structures and APIs.
- Fine-grained control over data and flow of logic is needed.

For example, the process of placing an order involves multiple steps (authorizing the credit card, processing payment, arranging delivery, etc.); these steps must be performed in a particular sequence. The most appropriate design approach for this kind of business process is to create business components to encapsulate each individual step in the process and to orchestrate those components using a business workflow.

Designing Business Components

Business components can be the root of atomic transactions. They implement business rules in diverse patterns and accept and return simple or complex data structures. Business components should expose functionality in a way that is agnostic to the data stores and services needed to perform the work, and should be composed in meaningful and transactionally consistent ways.

Business logic will usually evolve and grow, providing higher-level operations and logic that encapsulates pre-existing logic. In many cases, it is necessary to compose pre-existing business functionality in order to perform the required business logic. When composing business logic, take special care when transactions are involved.

If the business process will be invoking other business processes in the context of an atomic transaction, all the invoked business processes must ensure their operations participate in the existing transaction so that their operations will roll back if the calling business logic aborts. It should be safe to retry any atomic operation if it fails without fear of making data inconsistent. Conceptually a transaction boundary can be also though of as a retry boundary. Transactions across servers running Windows can be managed using Distributed Transaction Coordinator (DTC), which is used by .NET Enterprise Services. To manage distributed transactions in heterogeneous environments, use COM Transaction

Integrator (COMTI) and Host Integration Server 2000. For more information about COMTI and Host Integration Server, see

http://www.microsoft.com/hiserver.

If implementation of atomic transactions it is not possible, then use compensating methods and processes. Note that a compensating action does not necessarily roll back all application data to the previous state, but rather restores the business data to a consistent state. This is an important distinction; one that must be considered during the design process. For example, a supplier may expose a B2B shopping interface to partners. A compensating action for canceling an order being processed may involve charging an order cancellation fee. For long-running transactions and processes, the compensating action may be different at different states in the workflow, design these for appropriate stages in the process.

For information about handling transactions and isolation level issues, see "Transactions" in ".NET Data Access Architecture Guide" on MSDN

(http://msdn.microsoft.com/library/en-us/dnbda/html/daag.asp).

The following list summarizes the recommendations for designing business components:

- Rely on loosely coupled message-based communication that leverages web services as much as possible.
- Ensure that processes exposed through service interfaces will not reach an inconsistent state if the same message is received twice.
- Choose transaction boundaries carefully so that retries and composition are possible. This applies to both atomic and long-running transactions. Also consider using retries for message-based systems, especially when exposing the application functionality as a service.
- Business components should be able to run as much as possible in the context of any service user—not necessarily impersonating a specific application user. This provides the ability to invoke them with mechanisms that do not transmit or delegate user identity.
- Choose and keep a consistent data format (such as XML, DataSet, etc.) for input parameters and return values.
- Set transaction isolation levels appropriately. For information about handling transactions and isolation level issues, see "Transactions" in ".NET Data Access Architecture Guide" on MSDN (http://msdn.microsoft.com/library/en-us/dnbda/html/daag.asp).

Implementing Business Components with .NET

Create components that encapsulate business logic using the .NET Framework. Managed code can take advantage of Enterprise Services for distributed transactions and other services commonly needed in distributed applications.

Business component characteristics:

- Are invoked by the user process tier, service interfaces, and other business processes, typically with some business data to operate on, expressed as a complex data structure (a document).
- Are the root of transactions and therefore must vote in the transactions they participate in.
- Should validate input and output.
- May expose compensating operations for the business processes they provide.
- May call data access logic components to retrieve and/or update application data.
- May call external services through service agents.
- May call other business components and initiate business workflows.
- May raise an exception to the caller if something goes wrong when dealing with atomic transactions.
- May use the features of Enterprise Services for initiating and voting on heterogeneous transactions. Consider the fact that different transaction options can have a great impact on performance. However, transaction management is not an adjustment mechanism or variable for improving application performance. For performance comparisons of different transaction approaches, see "Performance Comparison: Transaction Control" on MSDN (http://msdn.microsoft.com/library/en-us/Dnbda/html/Bdadotnetarch13.asp). Transactional settings can be:
    - **Required**. Use this option for components that may be the root of a transaction, or that will participate in existing transactions.
    - **Supported**. Use this option for components that do not necessarily require a transaction, but that have been designed to participate in an existing transaction if one exists.
    - **RequiresNew**. Use this option when the component is used to start a new transaction that is independent of existing transactions.

Note:
*Using the **RequiresNew** and **NotSupported** options will affect transaction composability, be aware of the impact of retrying a parent transaction.*

- **NotSupported**. Use this option to prevent the component from participating in transactions.

Business components are called by the following consumers:

- Service interfaces
- User process components
- Business workflows
- Other business components

Figure 8 shows a typical business component interacting with data access logic components, service interfaces, service agents, and other business components.

Note:
The arrows in Figure 1.7 represent control flow, not data flow.

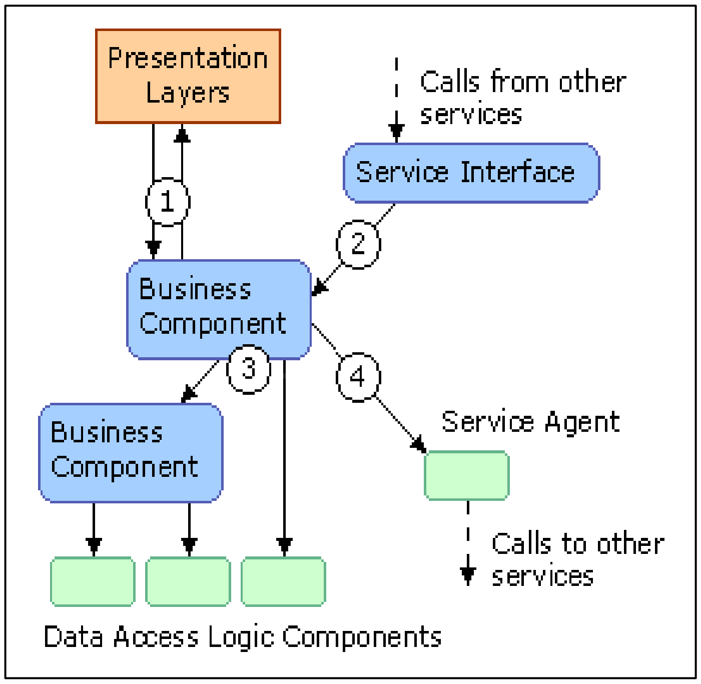


Figure 8 – Business Component Interaction

Note the following points in Figure 8:

1. Business components can be invoked by components in the presentation tiers (typically user process components) or by business workflows (not shown).

2. Business components can also be invoked by service interfaces (for example, an XML Web service or a Message Queuing listener function.

3. Business components call data access logic components to retrieve and update data, and they can also invoke other business components.

4. Business components can also invoke service agents. Take extra care in designing compensation logic in case the service being accessed is unavailable or takes a long time to return a response.

When to Use Enterprise Services for Business Components

Enterprise Services is the obvious choice for a host environment for business components. Enterprise Services provide the components with role-based security, heterogeneous transaction control, object pooling, and message-based interfaces for components by means of Queued Components (among other things). Enterprise Services do not have to be utilized in an application, however for anything more than simple operations against a single data source, its services will be needed, and taking advantage of the model provided by Enterprise Services early on, provides a smoother growth path for the system.

Decide at the very beginning of the design process whether or not to use Enterprise Services when implementing business components, because it will be more difficult to add or remove Enterprise Services features from the component design and code after it is built.

When implementing components with Enterprise Services, be aware of the following design characteristics:

- **Remote channel restriction**. Only HTTP and DCOM-RPC channels are supported. For more information, see “Designing the Communications Policy” in the, "Security, Operational Management, and Communications Policies."

- **Strong-named components**: Sign these components and all components they use in turn.

- **Deployment**. Components will either be self-registering (in which case they will require administrative rights at run time), or it will be necessary to perform a special deployment step. However, most server-side components require extra deployment steps anyway (to register Event Log sources, create Message Queuing queues, etc.).

- **Security**. Choose whether to use the Enterprise Services role model, who is based on Windows authentication, or to use .NET-based security.

For more information about Enterprise Services, see "Understanding Enterprise Services in. NET" on MSDN

(http://msdn.microsoft.com/library/en-us/dndotnet/html/entserv.asp).

Commonly Used Patterns for Business Components

Regardless of whether business components are hosted in Enterprise Services, there are many common patterns for implementing business tasks in the code. Commonly used patterns include:

- **Pipeline pattern**. Actions and queries are executed on a component in a sequential manner.

  A pipeline is a definition of steps that are executed to perform a business function. All steps are executed sequentially. Each step may involve reading or writing to data confirming the "pipeline state," and may or may not access an external service. When invoking an asynchronous service as part of a step, a pipeline can wait until a response is returned (if a response is expected), or proceed to the next step in the pipeline if the response is not required in order to continue processing.

  Use the pipeline pattern when:

  - The sequence of a known set of steps can be specified.
  - It is not necessary to wait for an asynchronous response from each step.
  - All downstream components must be able to inspect and act on data that comes from upstream (but not vice versa).

  Advantages of the pipeline pattern include:

  - It is simple to understand and implement.
  - It enforces sequential processing, which is good for certain types of processing.
  - It is easy to wrap in an atomic transaction.

  Disadvantages of the pipeline pattern include:

  - The pattern may be too simplistic, especially for service orchestration in which it may be necessary to branch the execution of the business logic in complex ways.
  - It does not handle conditional constructs, loops, and other flow control logic well. Adding one step affects the performance of every execution of the pipeline.

  The pipeline pattern is used extensively in applications based on Microsoft Commerce Server. For more information about how pipelines are used with

Commerce Server, see "Pipeline Programming Concepts" in the Commerce Server 2000 SDK documentation on MSDN

(http://msdn.microsoft.com/library/en-us/comsrv2k/htm/cs_sp_pipelineobj_woce.asp).

- **Event pattern**. Events are fired under particular business conditions, and code is written to respond to those events.

The event pattern may be used to have many activities happen but all receive the same starting data and cannot communicate with each other. Activities may execute in parallel or sequentially. Different implementations of the event may or may not run, depending on specific filtering information. If the implementations are set to run sequentially, order cannot be guaranteed.

Use the event pattern when:

- Managing independent and isolated implementations of a specific 'function' independently.
- Responses from one implementation do not affect the way another implementation works.
- All implementations are write only or fire-and-forget, where the output of the business process is defined by none of the implementations, or by just one specific business implementation.

Advantages of the event pattern include:

- Maintainability is improved by keeping unrelated business process independent.
- It encourages parallel processing, which may result in performance benefits.
- It is easy to wrap in an atomic transaction.
- It is agnostic to whether implementations run asynchronously or synchronously because no reply is expected.

Disadvantages of the event pattern include:

- It does not enable building complex responses for the business function.
- A component cannot use the data or status of another component in the event pattern to perform its work.

Enterprise Services provides the Events service, which provides a good starting point implementation of the event pattern. For more information about Enterprise Services Events, see "COM+ Events" in the COM+ SDK documentation on MSDN (http://msdn.microsoft.com/library/en-us/cossdk/htm/pgservices_events_2y9f.asp).

Implementing Business Workflows with BizTalk Server

When business processes require multiple steps or long-running transactions, manage the workflow, handling conversation state and exchanging messages with diverse services as required. BizTalk Server includes orchestration services that help meet these challenges.

Design business processes using BizTalk Server Orchestration services, and create XLANG schedules that implement business functionality. XLANG schedules are created graphically using BizTalk Server Orchestration Designer and can use BizTalk Messaging Services, .NET components, COM components, Message Queuing, or script to perform business tasks. XLANG schedules can be used to implement long-running transactions, and they automatically persist their state in a SQL Server database.

BizTalk Server Orchestration can be used to implement most kinds of business functionality. However, it is particularly suitable when the business process involves long-running workflow processes in which business documents are exchanged between multiple services. Documents can be submitted to BizTalk Server programmatically, or they can be delivered to a monitored file system folder or message queue known as a receive function. Receive functions ensure that the delivered documents match the specification defined for expected business documents, and if so, they consume the document and submit it to the appropriate business process channel in BizTalk Server. From this point of view, a receive function can be thought of as a simple form of service interface.

For an in-depth example that shows how to implement a business process using BizTalk Server Orchestration and Visual Studio .NET, see "Building a Scalable Business Process Automation Engine Using BizTalk Server 2002 and Visual Studio .NET" on MSDN (http://msdn.microsoft.com/library/en-us/dnbiz2k2/html/BizTalkVSautoeng.asp).

When the business process involves interactions with existing systems, such as mainframe applications, BizTalk Server can use adapters to integrate with them. For more information about integrating BizTalk Server with existing systems, see "Legacy File Integration Using Microsoft BizTalk Server 2000" on MSDN (http://msdn.microsoft.com/library/en-us/dnbiz/html/legacyfileint.asp).

BizTalk Server Orchestration Implementation

Figure 9 shows how an orchestrated business process interacts with service interfaces, service agents, and business components.

Note the following points in Figure 9:

1. Business workflows can be invoked from other services or from the presentation components (usually from user process components) using the service interface.
2. A business workflow invokes other services through a service agent, or directly through the service interfaces. Every outgoing message does not necessarily need to match an incoming message. Implement service interfaces and service agents in code, or if only simple operations are required, use the message transformation and functoid features of BizTalk Server.
3. Business workflows invoke business components. The business workflow or the components that it invokes can initiate atomic transactions.
4. Business workflows invoke data access logic components to perform data-related activities.
5. When designing business workflows, consider long response times, or method invocations with no reply at all. BizTalk Server automatically allows for long running conversations with external services.

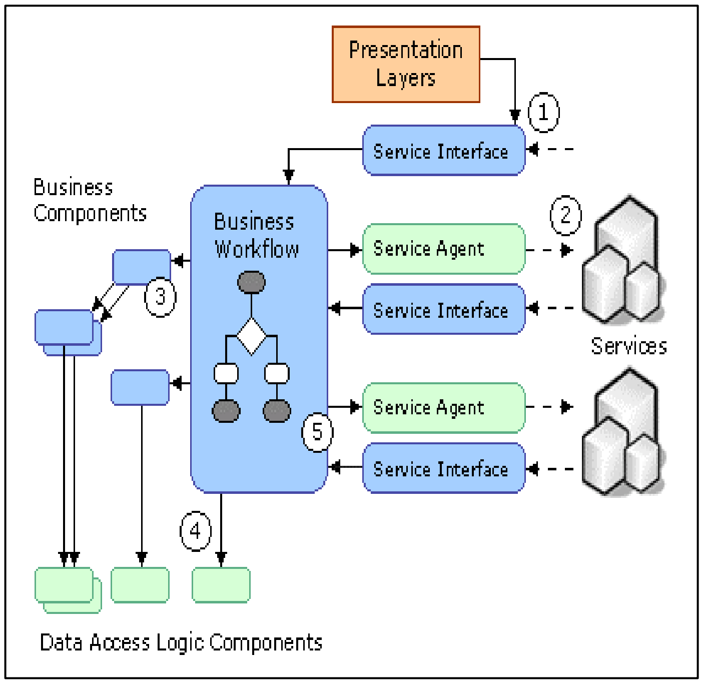


Figure 9 – An orchestrated business process

BizTalk Server Orchestration schedules are created graphically using the BizTalk Server Orchestration Designer. Figure 10 shows how an orchestration flow in the previous figure would look as rendered by Microsoft Visio® drawing and diagramming software. Notice how similar the conceptual diagram in Figure 10 looks to the flow a business analyst and developer needs to work with.

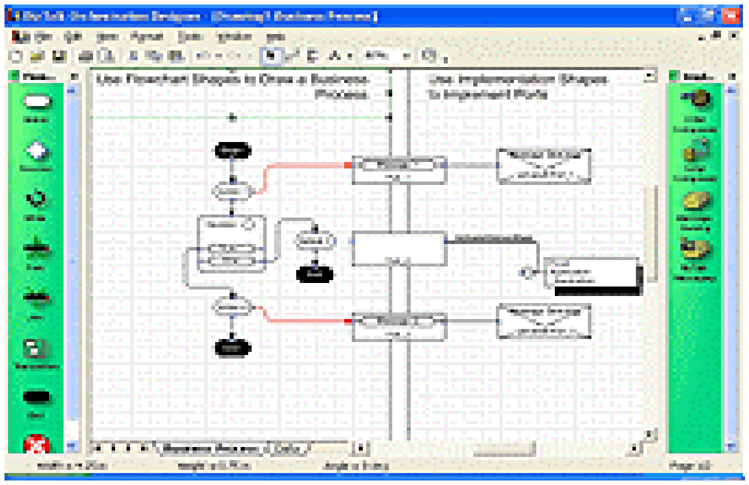

Use Flowchart Shapes to Draw a Business Process
Use Implementation Shapes To Implement Ports

Figure 10 – An orchestration flow in BizTalk Server Orchestration Designer

The drawing is then compiled into an XLANG schedule, which is an XML format file containing the instructions necessary for BizTalk Server to perform the tasks in the business process.

After it is compiled, the schedule can be initiated in one of the following ways:

- A BizTalk Server message can be submitted to BizTalk Server programmatically or through a file system or Message Queuing receive function.
- A schedule can be started programmatically using the *sked* moniker.

For more information about BizTalk Server Orchestration, read *BizTalk Server: The Complete Reference* by David Lowe et al (published by Osborne/McGraw Hill) and "Designing BizTalk Orchestrations" in the BizTalk Server 2000 documentation (http://msdn.microsoft.com/library/en-us/biztalks/htm/lat_sched_intro_xiju.asp).

For information about adapters for BizTalk:
http://www.microsoft.com/biztalk/evaluation/adapters/adapterslist.asp

The BizTalk Server Adapter's Developer Guide can be found at:
http://www.microsoft.com/biztalk/techinfo/development/wp_adapterdevelopersguide.asp

Designing a Service Interface

If exposing business functionality as a service, provide an entry point for clients to call that abstracts the internal implementation. It may also be necessary to expose similar functionality to different callers with different authentication requirements and service level agreement (SLA) commitments. Provide an entry point to the service by creating a service interface.

A service interface is a software entity typically implemented as a façade that handles mapping and transformation services to allow communication with a service, and enforces a process and a policy for communication. A service interface exposes methods, which may be called individually or in a specific sequence to form a conversation that implements a business task. For example, the Common Payment Service - credit card processing service in the application scenario might provide a method named AuthorizeCard that verifies credit card details, and a second method named ProcessPayment that transfers funds from the cardholder's account to the retailer. These steps would be performed in the appropriate sequence to process an order payment

The necessary communication format, data schema, security requirements, and process are determined as part of a contract, which is published by the service. This contract provides the information clients need to locate and communicate with the service interface.

When designing service interfaces, consider the following:

- Think of a service interface as a trust boundary for the application.
- If the service interfaces are exposed to external organizations and consumers, or made publicly available, design them in such a way that changes to an internal implementation will not require a change to the service interface.
- The same business logic in the service may need to be consumed in different ways by different clients, so it may be necessary to publish multiple service interfaces for the same functionality.
- Different service interfaces may define different communication channels, message formats, authentication mechanisms, performance service level agreements, and transactional capabilities. Common service level agreements are defined in time to respond to a certain request with a certain amount of information.

Implement service interfaces in different ways, depending on the functionality of the application or service is exposed:

- To expose business logic as an XML Web service, use ASP.NET Web service pages or expose some components through .NET remotely using SOAP and HTTP.

- To expose a service's functionality to clients sending Message Queuing messages, use Message Queuing Triggers or Enterprise Services Queued Components, or write custom 'message receiving' services.

For more information, see “Designing the Communications Policy” in the, "Security, Operational Management, and Communications Policies."

Service Interface Characteristics

Consider the following design characteristics of service interfaces:

- Sometimes the .NET infrastructure will allow the use of a transparent service interface (for example, Enterprise Services objects may be exposed as Web services in Windows Server 2003), and sometimes it may be necessary to add specific artifacts to the application, such as XML Web services, BizTalk Orchestration workflows, or messaging ports. Consider the impact of using transparent service interfaces, because they may not provide the abstraction necessary to facilitate changes to the business functionality at a later date without affecting the service interface. Implementing façades has its development cost, but will help isolate changes and to make the application more maintainable.
- Service interfaces can implement caching, mapping, and simple format and schema transformations; however, these façades should not implement business logic.
- The service interface may involve a transactional transport (for example, Message Queuing) or a non-transactional transport (for example, XML Web services over HTTP). This will affect the error and transaction management strategy.
- Design service interfaces for maximum interoperability with other platforms and services, relying whenever possible on industry standards for communications, security, and formats, standard or simple message formats (for example, simple XML schemas for XML Web services), and non-platform specific authentication mechanisms.
- Sometimes the service interface will have a security identity of its own, and will authenticate the incoming messages but will not be able to impersonate them. Consider using this approach when calling business components that are deployed on a different server from the service interface.

Using Business Facades with Service Interfaces

The channel or communication mechanism used to expose business logic as a service may have an associated way of implementing the service interface code. For example, if it is chosen to build Web services, most of the service interface logic will reside in the Web service itself, namely the asmx.cs files. Exposing the service through Message Queuing enables use of Queued Components from Enterprise Services, custom listeners, or Message Queuing Triggers to "fire up" the component that acts as service interface.

When planning to build a system that may be invoked through different mechanisms, add a façade between the business logic and the service interface. By implementing this façade, it is possible to consolidate in one place policy-related code (such as authorization, auditing, validations, etc.) so it can be reused across multiple service interfaces that deal with diverse channels. This façade provides extra maintainability because it isolates changes in the communication mechanisms from the implementation of the business components. The service interface code then only deals with the specifics of the communication mechanism or channel (for example, examining Web service SOAP headers or getting information from Message Queuing messages) and sets the proper context for invoking the business façade component. Figure 11 shows a business façade used in this manner.

Figure 11 (below) shows an example of how a business façade is used with the service interfaces of a system. IIS and ASP.NET receive an HTTP call (1) and invoke a Web service interface named *MyWebService.asmx* (2). This service interface inspects some SOAP message headers, and sets the correct principal object based on the authentication of the Web service. It then invokes a business façade component (3) that validates, authorizes, and audits the call. The façade then invokes a business component that performs the business work (4). Later the system is required to support Message Queuing, so a custom listener is built (5) that picks up messages and invokes a service interface component named *MyMSMQWorker* (6). This service interface component extracts data off the Message Queuing message properties (such as Body, Label, etc.) and also sets the correct principal object on the thread based on the Message Queuing message signature. It then invokes the business façade. By factoring the code of the business façade out of the service interface, the application was able to add a communication mechanism with much less effort.

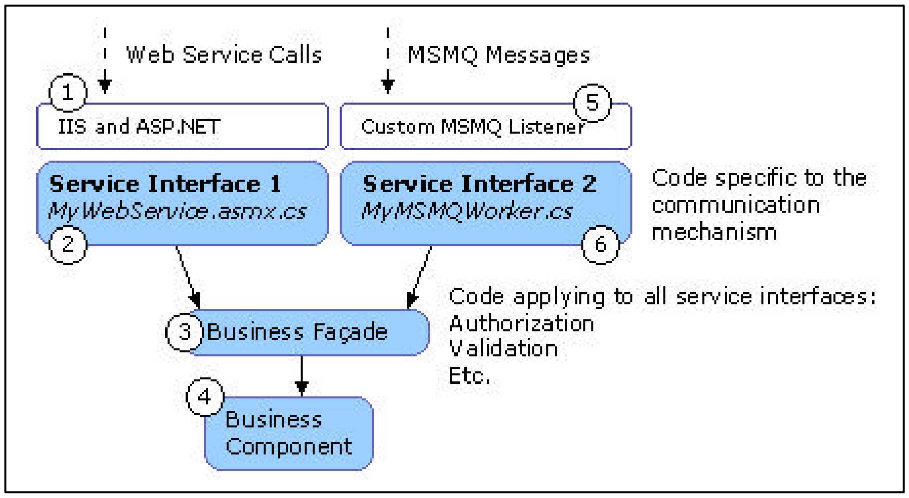


Figure 11 – Using a business façade with service interfaces

Transaction Management in Service Interfaces

The service interface must deal with a channel that provides transactional capabilities (such as Message Queuing) or one that doesn't (such as XML Web services). It is very important to design transaction boundaries so that operations can be retried in face of an error. To do so, make sure that all the resources used are transactional, mark the root component as "requires transaction," and mark all sub components as either "requires transaction" or "supports transactions."

With transactional messaging mechanisms, the service interface starts the transaction first and then picks up the message. If the transaction rolls back, the message is automatically "unreceived" and is placed back in the queue for a retry. When using Message Queuing, Enterprise Services Queued Components, or Message Queuing Triggers, define a message queue-and-receive operation as transactional to achieve this automatically.

When using a messaging mechanism that is not transactional (such as XML Web services), call the root of the transaction from the code in the service interface. In the case of a failure, design the service interface code to retry the operation or return to the caller an appropriate exception or preset data representing a failure.

Representing Data and Passing It Through Tiers

When the data access logic components return data, they can do so in a number of formats. These formats can vary from the data-centric (for example, an XML string) to the more object oriented (for example, a custom component that encapsulates an instance of a business entity). Common formats for returning data are:

- XML
- DataReader
- DataSet
- Typed DataSet
- Custom object with properties that map to data fields, and methods that perform data modifications through data access logic components.

For more information about the choices of data formats available in the application design, see "Designing Data Tier Components and Passing Data Through Tiers" on MSDN

(http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/BOAGag.asp?frame=true).

The data format chosen depends on how the data is to be accessed. It is recommended to avoid designs requiring transfer data in a custom object-oriented format, because doing so requires custom serialization implementation and can create a performance overhead. Generally, a more data-centric format should be used, such as a DataSet, to pass the data from the data access logic components to the business tiers, and then use it to hydrate a custom business entity to work with the data in an object-oriented fashion. In many cases, though, it will be simpler just to work with the business data in a DataSet.

Representing Data with Custom Business Entity Components

In most cases, data should be worked with directly by using ADO.NET datasets or XML documents. This passes structured data between the tiers of the application without having to write any custom code. However, to encapsulate all the details about working with a particular format, or to add behaviors to data, it may be necessary to develop custom components. This provides tight control over what other application components can do with the data, enables abstraction of internal formats from the data schema that the application uses, and allows addition of behavior to the data. This Implementation Guide refers to the components used to represent data as *business entities*.

For example, the ordering process discussed earlier in this Implementation Guide could use an Order object, which has an associated Customer object, and a collection of Line Item objects. These components form part of the business tiers of the application, and can be consumed by other business components or by presentation components.

Note:
Custom business entity components are not a mandatory part of all applications. Many solutions (especially ASP.NET-based applications and business components) do not use custom representations of business entities, but instead use DataSets or XML documents because they provide all the required information and the development model is more task- and document- based as opposed to object-oriented.

Entity components contain snapshot data. They are effectively a local cache of information, so the data can only be guaranteed to be consistent if it is read in the context of an active transaction. Do not map one business entity to each database table; typically a business entity will have a schema that is a denormalization of underlying schemas. Note that the entity may represent data that has been aggregated from many sources.

Because the component stores data values and exposes them through its properties, it provides stateful programmatic access to the business data and related functionality. Avoid designing business entity components in such a way that the data store is accessed each time a property changes and should instead provide an Update method that propagates all local changes back to the database. Business entity components should not access the database directly, but should use data access logic components to perform data-related work as their methods are called. Business entities should not initiate any kind of transactions, and should not use data access APIs—they are just a representation of data, potentially with behavior. Because they may be called from business components as well as user interfaces, they should flow transactions transparently and should not vote on any ongoing transaction.

Design business entity components to be serialized to persist current state (for example, to store on a local disk if working offline, or into a Message Queuing message).

Business entity components simplify the transition between object-oriented programming and document-based development models. Object-oriented design is common in stateful environments such as user interface design, whereas business functionality and transactions can typically be expressed more clearly in terms of document exchanges.

Business Entity Component Interface Design

Business entity components expose:

- Property accessors (get and set functions) for attributes of the entity.
- Collection accessors for sub collections of related data. (The collections don't necessarily yield collections of business entities, design the service entity to expose Datasets or Data Tables directly and not be concerned about object model traversal.)
- Control functions and properties commonly used in entity management, for example, Load, Save, IsDirty, and Validate.
- Methods to access metadata for the entity, which can be useful in improving maintainability of the user interface.
- Events to signal changes in the underlying data.
- Methods to perform business tasks or get data for complex queries. These methods may act on the local data only (for example, Order.GetTotalCost) or on the business components and processes (for example, Order.Place).
- Methods and interfaces needed for data binding.

Consumers of business entity components include:

- User interaction components for rich clients. These components may bind to the data in business entities or the data exposed by any queries the component may expose. UI controller functions may also set and get properties of business entities for data input and display.
- User process components. User process components may hold one or more business entities as part of their internal business-specific state.
- Business components. Business processes may pass a business entity as a parameter to a data access logic component method (for example, an Order object could be passed to an InsertOrder method in a data access logic component). Alternatively, business components could also use business

entities to access data behavior (for example by calling a Place method on the Order object, which in turn passes the order data to a data access logic component), but this approach is more uncommon than passing the business entity directly to a data access logic component method because it mixes a functional, document-oriented model with an object-based model.

Recommendations for Business Entity Design

These recommendations will help to implement the right mechanism to represent the data:

- Carefully consider whether custom entity coding is needed or whether other data representations satisfy the requirements. Coding custom entities is a complex task that increases in development cost with the number of features it provides. Typically, custom entities are implemented for applications that need to expose a custom macro or a developer-friendly scripting object model for customization.
- Implement business entities by deriving them from a base class that provides boilerplate functionality and encapsulates common tasks.
- Rely on keeping internal datasets or XML documents for complex data instead of internal collections, structures, etc.
- Implement a common set of interfaces across business entities that expose common sets of functionality:
    - Control methods and properties, such as Save, Load, Delete, IsDirty, and Validate.
    - Metadata methods, such as getAttributesMetadata, getChildDatasetsMetadata, and getRelatedEntitiesMetadata. This is especially useful for user interface design.
- Isolate validation rules as metadata, for example by exposing XML Schema Definition Language (XSD) schemas. Make sure, however, that external callers cannot tamper with these validation rules.
- Business entities should validate the data they encapsulate through the enforcement of continuous and point-in-time validation rules.
- Implement an implicit enforcement of relationships between entities based on the data schema and the business rules around the data. For example, an Order object could have a maximum number of Line Item references.
- Design business entities to rely on data access logic components for database interaction. Doing so enables implementation of all data access policies and related business logic in one place. If the business entities access SQL Server databases directly, it will mean that applications

deployed to clients that use the business entities will need SQL connectivity and logon permissions.

For detailed design recommendations and sample code to assist when developing business entity components, see "Designing Data Tier Components and Passing Data Through Tiers" on MSDN

(http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/BOAGag.asp?frame=true).

Designing Data Tiers

Almost all applications and services need to store and access some kind of data. For example, the application discussed in this guide needs to store product, customer, and order data.

When working with data, determine:

- The data store being used.
- The design of the components used to access the data store.
- The format of the data passed between components, and the programming model it requires.

The application or service may have one or more data sources, and these data sources may be of different types. The logic used to access data in a data source will be encapsulated in *data access logic components*, which provide methods for querying and updating data. The data that the application logic requires to work is related to real-world *entities* that play a part in a business. In some scenarios, custom components representing these entities may be developed, while in others ADO.NET datasets or XML documents may be used directly.

Figure 12 shows how the logical data tier of an application consists of one or more data stores, and depicts a tier of data access logic components that are used to retrieve and manipulate the data in those data stores.

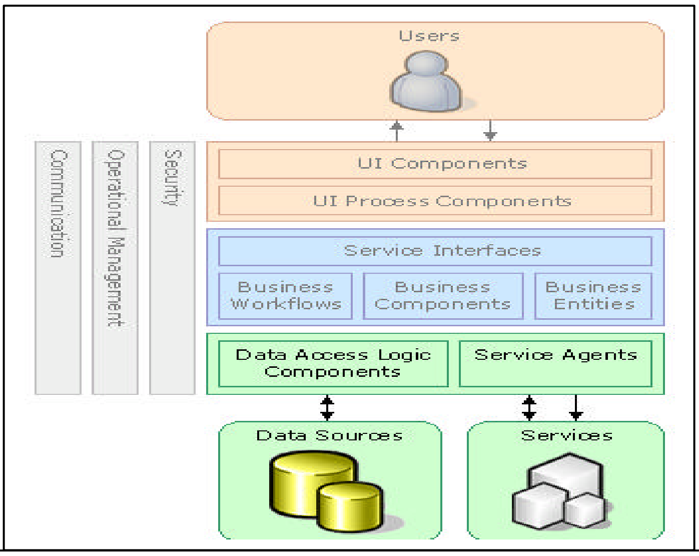


Figure 12 – Data components

Most applications use a relational database as the primary data store for application data. Other choices include legacy databases, the file system, or document management services.

When the application retrieves data from the database, it may do so using a data format such as a DataSet or DataReader. The data will then be transferred across the layers and tiers of the application and finally will be operated on by one of the components. Different data formats for retrieving, passing, and operating on the data; for example, the data in a DataSet to populate properties in a custom entity object might be used. However, strive to keep the formats consistent, because it will probably improve the performance and maintainability of the application to have only a limited set of formats, avoiding the need for extra translation layers and the need to learn different APIs.

Note:
As required by the STA, "stored procedures must only be used when necessary to improve performance." Additionally, stored procedures should be kept to a minimum because they "are database specific, they are not easily migrated of a new database is selected." Finally, stored procedures must never contain business loaic.

The following sections discuss the choice of data stores, the design of data access logic components, and the choices available for representing data.

Data Stores

Common types of data stores include:

- **Relational databases**. Relational databases such as Microsoft SQL Server 2000 provide high volume, transactional, high performance data management with security, operations, and data transformation capabilities. DSS and OLAP services are available as well. Relational databases also host complex data logic instructions and functions in the form of stored procedures that can be used as an efficient environment for data-intensive business processes. Microsoft SQL Server also provides a desktop and palm-held device version that provides transparent implementations for data access logic components. This document will focus on outlining implementation guidelines that should be used for data access. Database design is beyond the scope of this guide, for detailed relational database design information, see "Database Design Considerations" in the Microsoft SQL Server 2000 SDK

  (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/createdb/cm_8_des_02_62ur.asp).

- **File system.** It may be decided to store data in the file system. These files could be in a custom format or in an XML format with a schema defined for the purposes of the application. These should only be created for temporary use and need to be encrypted via either EFS of other means if the data is confidential.

There are many other stores (such as XML databases, online analytical processing services, data warehousing databases, etc.) but they are beyond the scope of this guide.

Data Access Logic Components

Regardless of the data store chosen, the application or service will use data access logic components to access the data. These components abstract the semantics of the underlying data store and data access technology (such as ADO.NET), and provide a simple programmatic interface for retrieving and performing operations on data.

Data access logic components usually implement a stateless design pattern that separates the business processing from the data access logic. Each data access logic component typically provides methods to perform Create, Read, Update, and Delete (CRUD) operations relating to a specific business entity in the application (for example, order). The business processes may use these methods. Specific queries can be used by the user interface to render reference data (such as a list of valid credit card types).

Note:
Data access logic components are recommended for all applications that need to access business data (such as products, orders, and so on). However, other products and technologies may use databases to store their own operational data, without the need for custom data access logic components.

When an application contains multiple data access logic components, it can be useful to use a generic data access helper component to manage database connections, execute commands, or cache parameters. The data access logic components provide the logic required to access specific business data, while the generic data access helper utility component centralizes data access API development and data connection configuration, and helps to reduce code duplication. A well-designed data access helper component should have no negative impact on performance, and provides a central place for data access tuning and optimization. Microsoft provides the Data Access Application Block for .NET (http://msdn.microsoft.com/library/en-us/dnbda/html/daab-rm.asp), which can be used as a generic data access helper utility component in applications when using Microsoft SQL Server databases.

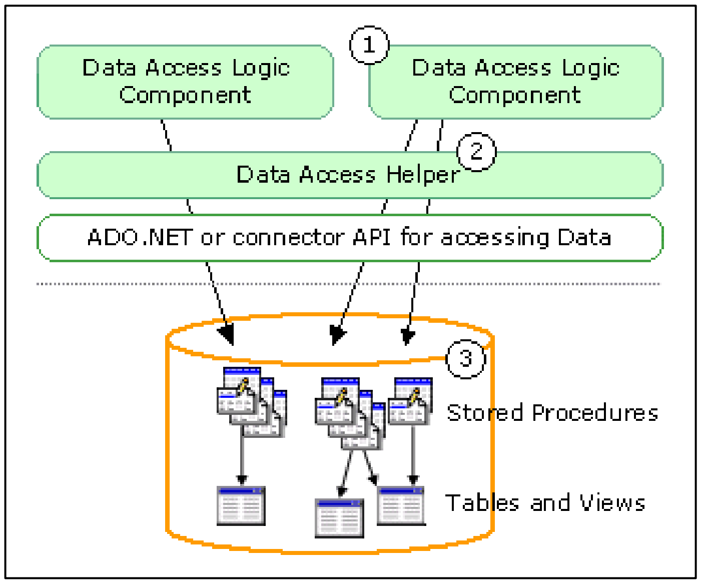

Data Access Logic Component
1
Data Access Logic Component
2
Data Access Helper
ADO.NET or connector API for accessing Data
3
Stored Procedures
Tables and Views

Figure 13 – Data access logic components

Note the following points in Figure 13:

1. Data access logic components expose methods for inserting, deleting, updating, and retrieving data. This includes the provision of paging functionality when retrieving large quantities of data.

2. A data access helper component can be used to centralize connection management and all code that deals with a specific data source.

3. Minimally, implement queries and data operations as stored procedures (if supported by the data source) to enhance performance and maintainability.

Data access logic components provide simple access to database functionality (queries and data operations), returning both simple and complex data structures. They hide invocation and format idiosyncrasies of the data store from the business components and user interfaces that consume them. Implementing data access logic in data access logic components enables encapsulation of all the data access logic for the entire application in a single, central location, making the application easier to maintain or extend.

Design each data access logic component to deal with only one data store. (This means that these components do not query and aggregate data from many sources; this is done by the business components.)

When using heterogeneous transactions, data access logic components should participate in them, but they must never be the root of the transaction. It is more appropriate to have a business component as the root of a transaction in which one or more data access logic components are used to perform database updates.

Data Access Logic Component Functionality

When called, data access logic components typically do the following:

- Perform simple mappings and transformations of input and output arguments. This abstracts business logic from database specific schemas and stored procedure signatures.

- Access data from only one data source. This improves maintainability by moving all data aggregation functionality to the business components, where data can be aggregated according to the specific business operation being performed.

- Act on a main table and perform operations on related tables as well. (Data access logic components should not necessarily encapsulate operations on just one table in an underlying data source.) This enhances the maintainability of the application.

Optionally, they may perform the following work:

- Use a custom utility component to manage and encapsulate optimistic locking schemes.
- Use a custom utility component to implement a data caching strategy for non-transactional query results.
- Implement dynamic data routing for very large-scale systems that provide scalability by distributing data across multiple database servers.

Data access logic components should not:

- Invoke other data access logic components. Avoiding a design in which data access logic components invoke other data access logic components helps keep the path to data predictable, thus improving application maintainability.
- Initiate heterogeneous transactions. Since each data access logic component deals with only a single data source, there will be no scenario in which a data access logic component is the root for a heterogonous transaction. In some cases, however, a data access logic component may control a transaction that involves multiple updates in a single data source.
- Maintain state between method calls.

Data Access Logic Component Interface Design

Data access logic components commonly need to provide an interface to the following consumers:

- **Business components and workflows**. Data access logic components need to provide I/O of disconnected business documents and/or scalars in stateless, functional style methods, such as GetOrderHeader().
- **User interface components**. The user interaction components may use data access logic components for I/O of disconnected business documents for rendering data in rich clients and disconnected client scenarios, or for streaming output (for example, obtaining a DataReader) for ASP.NET and clients that benefit from stream rendering. Consider using data access logic components directly from the user interface to take advantage of the faster

performance this design offers and if there is no need for additional business logic between the user interface and data source.

Data access logic components may connect to the database directly using a data access API such as ADO.NET, or in more complex applications it may be chosen to provide an additional data access helper component that abstracts the complexities of accessing the database. In either case, it may be advantageous to use stored procedures to perform the actual data retrieval or modification when using a relational database.

The methods exposed by a data access logic component may perform the following kinds of tasks:

- Common functionality that relates to managing "entities" such as CRUD functions.
- Queries that may involve getting data from many tables for read-only purposes. The data may be returned as paged or non-paged depending on requirements, and the results may be streamed or non-streamed depending on whether the caller can benefit from it.
- Actions that will update data and potentially also return data.
- Returning metadata related to entity schema, query parameters, and resultset schemas.
- Paging for user interfaces that require subsets of data, such as when scrolling through an extensive product list.

Input parameters to data access logic component methods will typically include scalar values and business documents represented by XML strings or DataSets. Return values may be scalars, DataSets, DataReaders, XML strings, or some other data format. For specific design and implementation guidance in choosing a data format for objects, see "Designing Data Tier Components and Passing Data Through Tiers" on MSDN

(http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/BOAGag.asp?frame=true).

Data Access Logic Component Example

The following C# code shows a partial skeleton outline of a simple data access logic component that could be used for accessing order data. This code is not intended to be a code template, but to illustrate some of the concepts from the discussion.

```
public class OrderData

{
 private string conn_string;
 public OrderData()
 {
   // acquire the connection string from a secure or encrypted
   // location and assign to conn_string
 }
 public DataSet RetrieveOrders()
 {
   // Code to retrieve a DataSet containing Orders data
 }
 public OrderDataSet RetrieveOrder(Guid OrderId)
 {
   // Code to return a typed DataSet named OrderDataSet
   // representing a specific order.
   // (OrderDataSet will have a schema that has been defined in Visual Studio)
 }
 public void UpdateOrder(DataSet updatedOrder)
 {
   // code to update the database based on the properties
   // of the Order data sent in as a parameter of type dataset
 }
}
```

Code Sample 1 – C# Simple Data Access Logic

Recommendations for Data Access Logic Component Design

When designing data access logic components, consider the following general recommendations:

- Return only the data needed. This improves performance and enhances scalability.
- Use stored procedures to abstract data access from the underlying data schema. *However, be careful not to overuse stored procedures, because doing so will severely impact the maintainability of the application in terms of code maintenance and reuse.* A symptom of overusing stored procedures is having large trees of stored procedures that call each other. Avoid using stored procedures to implement control flow, manipulate individual values (for example, perform string manipulation), or to implement any other functionality that is difficult to implement in Transact-SQL.
- Rely on RDBMS functionality for data-intensive work. Follow the principle, "Move the processing to the data, not the data to the processing." Balance using stored procedures against the maintainability and reusability of data logic.
- Implement a standard or expected set of stored procedures giving commonly used functionality, such as insert, read, update, and find functions. Doing so will save time when developing business components. Being proactive about implementing this functionality makes the implementations consistent and enforce internal standards. If the design seems to be repeatable, code generators can be used to build basic boilerplate stored procedures and data access logic component logic.
- Expose the expected functionality that is common across all data access logic components in a separately defined interface or base class.
- Design consistent interfaces for different clients:
    - Business components can be implemented in many ways, including the use of custom .NET code, BizTalk Orchestration rules, or a third-party business rule engine. The design of the interface for data access logic components should be compatible with the implementation requirements of current and potential business components to avoid having additional interfaces, façades, or mapping layers between both.
    - ASP.NET-based user interfaces will benefit in terms of performance from rendering data exposed as DataReaders. DataReaders are best for read-only, forward-only operations in which processing for each row is fast. If the data access logic components are deployed together with the user interface, expose large query results intended for rendering in data access logic component functions that return DataReaders. If planning to operate on the data for a longer period of time, scalability can be

improved by relying on a disconnected DataSet instead of a DataReader.

  - Have the data access logic components expose metadata (for example, schema and column titles) for the data and operations it deals with. Doing so can help make applications more flexible at run-time, especially when rendering data in user interfaces.

  - Do not build one data access logic component per table. Design data access logic components to represent a slightly higher level of abstraction and denormalization that is consumable from business processes. It is uncommon to expose a relationship table as such; instead, expose the relationship functionality as data operations on the related data access logic components. For example, in a database where a many-to-many relationship between books and authors is facilitated by a TitleAuthor table, do not create a data access logic component for TitleAuthor, but rather provide an AddBook method to an Author data access logic component or an AddAuthor method to a Book data access logic component. Semantically, add a book to an author or add an author to a book, but not "insert a uthorship."

- If encrypted data is stored, the data access logic components should perform the decryption (unless it is desired to encrypted the data to go all the way to the client).

- If hosting business components in Enterprise Services, build data access logic components as serviced components and deploy them in Enterprise Services as a library application. This allows them to participate and explicitly vote in Enterprise Services transactions and use role-based authorization. Data access logic components don't need to be hosted in Enterprise Services if none of the services are used or if they will be loaded in the same AppDomain as an Enterprise Services caller. For more information about using Enterprise Services, see "Business Components and Workflows".

- Enable transactions only when needed. Do not mark all data access logic components as Require Transactions, because this taxes resources and is unnecessary for read operations performed by the user interface. Instead, mark them as Supports Transactions by adding the following attribute:

- [Transaction (TransactionOption.Supported)]

- Consider tuning isolation levels for queries of data. If building an application with high throughput requirements, special data operations may be performed at lower isolation levels than the rest of the transaction.

Combining isolation levels can have a negative impact on data consistency, so carefully analyze this option on a case-by-case basis. Transaction isolation levels should usually be set only at the transaction root (that is, the business process components). For more information, see Designing Business Tiers earlier in this chapter.

- Use data access helper components. For benefits of this approach and details, see Designing Data Access Helper Components in this chapter.

For more information about designing data access logic components, see ".NET Data Access Architecture Guide"

(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp). Microsoft also provides the Data Access Application Block

(http://msdn.microsoft.com/library/en-us/dnbda/html/daab-rm.asp), a tested, high-performance data helper component that can be used in the application.

Designing Data Access Helper Components

When an application requires large numbers of data access logic components to access the same data source, it may be necessary to implement similar generic data access code in each data access logic component. This duplication of logic can lead to maintainability issues and makes it difficult to troubleshoot data access problems. Centralizing generic data access functionality in a data access helper component can produce a cleaner, more manageable design. Data access helper components provide an easy invocation model to the underlying data source. Consider data access helper components to be generic, caller-side façades into the data source. They are typically agnostic to the application business logic being performed. Usually only one or two helper components are needed for a given data source. Each one may implement different sets of technical functionality for accessing the service. For example, one data access helper component to a database may invoke stored procedures, while another one may stream large amounts of data out.

If designing the application to be agnostic to the data source type (for example, to be able to switch from an Oracle database to a SQL Server database), do so by having two simple data access helper components that expose a similar interface. It is important to understand that porting an application to a different database will require a varying level of effort. The important concept to put into practice here is to design the application in such a way to keep the effort required to port at a minimum.

The goal of using a data access helper component is to:

- Abstract the data access API programming model from the data-related business logic encapsulated in the data access logic components, thus reducing and simplifying the code in the data access logic components.

- Isolate connection management semantics.
- Isolate data source location (through connection string management).
- Isolate data source authentication.
- Isolate transaction enlistment (ADO.NET does this automatically when used to access data in a SQL Server database or when using ODBC or OLEDB).
- Centralize data access logic for easier maintenance, minimizing the need for data source-specific coding skills throughout the development team and making it easier to troubleshoot data access issues.
- Isolate data access API versioning dependencies from data access logic components.
- Provide a single point of interception for data access monitoring and testing.
- Use code access and user-based or role-based authorization to restrict access to the whole data source.
- Translate non-.NET exceptions that may be returned by the data source into exceptions that the application can handle in traditional ways.

To see an example of a data access helper component, including source code and documentation, download the Data Access Application Block for .NET from MSDN (http://msdn.microsoft.com/library/en-us/dnbda/html/daab-rm.asp).

Accessing Multiple Data Sources

When accessing an Oracle database or other data sources, it may be preferred to abstract as much as possible the API that is used to access them from the data access logic components. Microsoft has provided Oracle and OLE DB implementations of the Data Access Application Block and has stress-tested them in the context of the Nile performance benchmark. These implementations are available for download on MSDN by following the links in this article:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/manprooracperf.asp

Achieving RDBMS transparency is a complex design goal, and using data access helpers can help to mitigate some of the development, troubleshooting, and maintenance efforts. However, testing the application is still necessary with each data source due to the different ways in which relational database management systems handle stored procedures, cursors, locking, and other database artifacts.

If the application may be deployed in different environments with different relational database management systems, implement the data access helpers with a common interface and provide the actual component that does the data access for a particular data source in a factory pattern. Change the source code supplied for the Application Blocks for .NET mentioned earlier to accommodate these specific requirements.

Database Connectivity:

- For SQL Server 2000 and 7.0 databases the preferred data access method is the SQL Server .NET Data Provider for data access. It is faster and more efficient.
- For Oracle databases, it is recommended to use the Managed Provider for Oracle. This will achieve greater performance for data access with Oracle databases
- For DB2 and UDB databases, there is an upcoming managed provider that should be available in the mid 2003. This would be the preferred data access method for these types of database.
- For other OLE/DB compliant database – use the OLE/DB Connector and Data Provider. This allows consistent data access across OLE/DB compliant database such as Oracle, DB2 and Sybase.

Integrating with Services

If the business process involves external services, handle the semantics of communicating with each service called. Specifically, use the correct communication API to call the service and perform any necessary translation between the data formats used by the service and those used by the business process. If the service contract consists of a long-running conversation, keep intermediate state while waiting for a response.

Use a service agent component that encapsulates the logic necessary to encapsulate these tasks and to initiate and manage a messaging-based conversation for each service the application must consume. Think of service agents as data access logic components for services other than data stores, or as proxies or emissaries to other services. Some service publishers may provide callers with a ready-built service agent, while in other cases it may be necessary to custom develop.

The goal of using a service agent is to:

- Encapsulate access to one service.
- Isolate the business process implementation from the service implementation in terms of data format or schema changes.
- Provide input and output data formats that are compatible with the business components calling the service.

Service agents may also perform the following common type of tasks if required:

- Perform basic validation of the data exchanged with the service.
- Cache data for common queries.
- Authorize access to the service, providing a granular way to check security before accessing the service from the calling application's perspective. Typically, the service will authenticate and authorize requests as well.
- Set the right security context or provide the right credentials to the service for authentication. For example, to set the credentials for an XML Web service being invoked, use the HTTPCredentialCache.
- Make sure the right portions of the message are encrypted or that a secure channel can be established if necessary.
- Provide monitoring information that allows interaction with the service to be instrumented. This allows determination of whether partners are complying with their service level agreements (SLAs).

Managing Asynchronous Conversations with Services

In some cases, it is necessary to integrate the application with other services, both sending and receiving asynchronous calls. In this case, service interfaces will be receiving calls from the outside services, and calls will be made into those services from the service agents. If these message exchanges are implemented in an asynchronous way, it may be necessary to keep track of the conversation a certain set of message exchanges belong to. Use one of these two options to keep track of the conversation state:

- Use the business data in the messages to identify the conversation. For example, an order ID number may be used in all messages to identify the order being processing in a particular message exchange. This is the most straightforward way of correlating messages.
- Provide an infrastructure component or utility that generates GUIDs or IDs for specific conversations and attaches them to messages. Service agents and service interfaces will need access to this information to understand how to interpret a particular asynchronous call. A persistent database will also be necessary to track the state and ID of each conversation. This requires extra development, and the context of the message is lost if the message needs to be interpreted outside the service.

However, it may be convenient to use correlation IDs if that information is maintained private.

Language Adoption

There are over 20 languages available for Microsoft Visual Studio.NET. The intent is to make the transition easy for programmers coming from other languages. There are obviously pros and cons for having multilingual solutions. To add consistency and structure to the solution development process, it is recommended that the following be adhered to:

- Agencies should conform to no more than 1 of 2 languages – preferably 1.
- For the majority of the Microsoft developers in the State of North Carolina, Visual Basic.NET will be the preferred language.
- For Java developers, it is recommended that they adopt C# since it is very similar to Java and the transition should be painless.
- For COBOL and RPG programmers, the transition to Visual Basic.NET should be straightforward. They can purchase the language complier for the language they already know. This should be decided on a case-by-case basis.

Upgrading from Visual Basic 6.0

Deciding when to upgrade

This section deals with the upgrade roadmap for Visual Basic 6.0 applications. Upgrading existing applications is not a requirement but is recommended. Existing code in VB 6.0 may be left alone and minor modifications to the applications may continue in VB 6.0. Any new significant modifications should be done using the .NET framework. COM+ Interop should be used to interoperate between VB 6.0 and VB.NET. Overtime, existing legacy COM+ applications should be re-developed and re-deployed via the .NET framework. By adopting a remediation approach of upgrading applications gradually, the impact to the business can be minimized.

This approach, especially with the simple projects to start off with, gives developers a chance to familiarize themselves with the new VB.NET language, syntax and the environment. It may also be possible to upgrade only certain portions of a solution. For example, initially upgrade the UI, but leave the business rules and data tier as COM objects.

There are a number of considerations when it comes to upgrading to VB.NET. One of them is the application architecture and technologies, and the other is whether the solution is in maintenance, development or planning phase. Each project should be evaluated on a case-by-case basis.

There is detailed information in the help files for Visual Studio.NET to aid in the decision process. The intent of this document is to highlight upgrade concerns to be aware of and also provide guidance for the business decision for deciding when to upgrade.

In short, there are two types of issues - Micro and Macro. Micro issues are simple coding changes. For example, there is no App.Path in .NET, but with one line of code the application directory can still be determined. Macro issues are tougher to solve, and may require some re-design of small portions of the application.

Micro Issues:

- Some uses of variants may not upgrade seamlessly. .NET is much stronger typed than VB6. It may be necessary to write code to convert the variant to the appropriate type.
- .NET doesn't support default properties. In most cases, the wizard will determine the default property, and upgrade it appropriately. For example, "Text1 = Something" will be upgraded to "Text1.Text = Something". However, if late binding is being used, this is not possible since late bound objects are "typeless" at compile time.
- There is not an "App" object in .NET, but its functionality is available through other classes. Some of these issues will be handled automatically by the upgrade wizard. Others will require hand editing.
- If using user defined types, they generally upgrade without a problem. However, if using UDTs that contain fixed length strings, or arrays, then code must be written that will initialize these elements, since this is now done at runtime, not by the compiler.
- .NET does not support the "As Any" variable type. This was often used to pass a variable that was either a string, or null, to an API function. In .NET, overloads are made to have a function signature that accepts the correct type.
- In .NET, all arrays are zero based. If code uses arrays that are not zero based, then changes are needed often.
- Other legacy keywords have simply been removed. For example, LSet may be used to copy the contents of one UDT to another. In .NET it is necessary to implement a custom method to perform this operation.

Macro Issues:

For the micro issues, it's simply a matter of writing slightly different code to accomplish the same task. For Macro issues, it must be decided whether it's worth upgrading, or to remain on VB6. Some of these issues require architectural changes to solve.

- ActiveX documents and DHTML applications don't automatically upgrade to VB6. However, it is possible to navigate from an ActiveX documents and DHTML applications to an ASP.NET page, and back. Since these applications interoperate with .NET, they may be left alone.
- StrPtr, ObjPtr, and VarPtr were undocumented functions that provided the memory address of variables. This kind of operation is considered "unsafe" in .NET. If this functionality is needed (and usually it is not), then use C# to pin objects in memory, and write "unsafe" code (code that is not managed by the Common Language Runtime) to access their memory.
- The graphics API has changed to GDI+. It does everything it did before, and more, but it's a different set of classes.
- .NET handles the lifetime of objects differently. If a lot of code was placed in the "Terminate" method of a class, this will have to be moved to a "Dispose" method. It is also necessary to modify code to call Dispose.
- DAO and RDO can still be used from .NET, but it is not possible to databind .NET classes to these objects. Data access code should be upgraded to ADO.NET
- .NET offers rich support for COM+ transactions. This is provided through the Enterprise Services namespace.
- Rather than using the legacy of On x Goto, construct code with select case statements.

Reasons for upgrading:

- Shorten the time that it takes to make enhancements.
- .NET reduces deployment costs by supporting an XCopy deployment model.
- Better tools for tracing and debugging that can significantly cut maintenance costs.

- VB.NET is now an object-oriented language offering many advantages:
    - Inheritance
    - Polymorphism
    - Structured Exception Handling
    - Overloading
    - Overriding
    - Constructors and Destructors

Also, pure .NET applications deploy much easier. DLLs don't register! Installing an application is largely a process of copying the files (this assumes that the application is 100% managed code. If the application is dependent on COM objects or ActiveX controls, a Windows Installer package must be built). Uninstalling is just deleting the files. To make this even easier, and to bundle the .NET runtime with the application, VS.NET enables building .msi and .cab deployment projects.

Interoperation with Java Based Applications

Interoperation with Java based applications should be done using web services. Since one of the main benefits of Microsoft.NET and web services is interoperability, this should be one of the main benefits for the adoption of web services. Since web services are based on standards such as SOAP, WSDL and UDDI, a web service created in Microsoft.NET will be able to interface with a web service created in a Java EJB environment. This does imply that the backend logic will need to be exposed on the Java side through SOAP and WSDL. There are tools on that platform to accomplish this.

There is also java to .NET bridge software that builds proxy classes in whatever environment desired; there are products available that accomplish this. For support with this approach, please consult the corresponding documentation.

Interoperation with COM+ and Enterprise Services

Not all code will be re-written at native .NET code overnight. In fact, many Microsoft products will carry COM+ objects for quite some time. Therefore, it's critical that .NET code can interface with COM+ code, and that COM+ code can talk to .NET code. Interop preserves the existing investment in COM+. Interop allows migration to .NET one piece at a time. The whole application does not need to be ported at once; it can be worked on project-by-project.

Authentication for the State of North Carolina

The Identity and Access Management Service (IAMS) was established to fulfill STA requirements by providing a common centralized shared service to authenticate, authorize, and manage the identities of users as well as provide audit logging of user activity. Agencies are required to utilize this service for all in-house developed applications.

IAMS is a Commercial-Off-The-Shelf (COTS) solution that may be customized to meet agency business requirements, as shown in the sample workflow in figure 14.

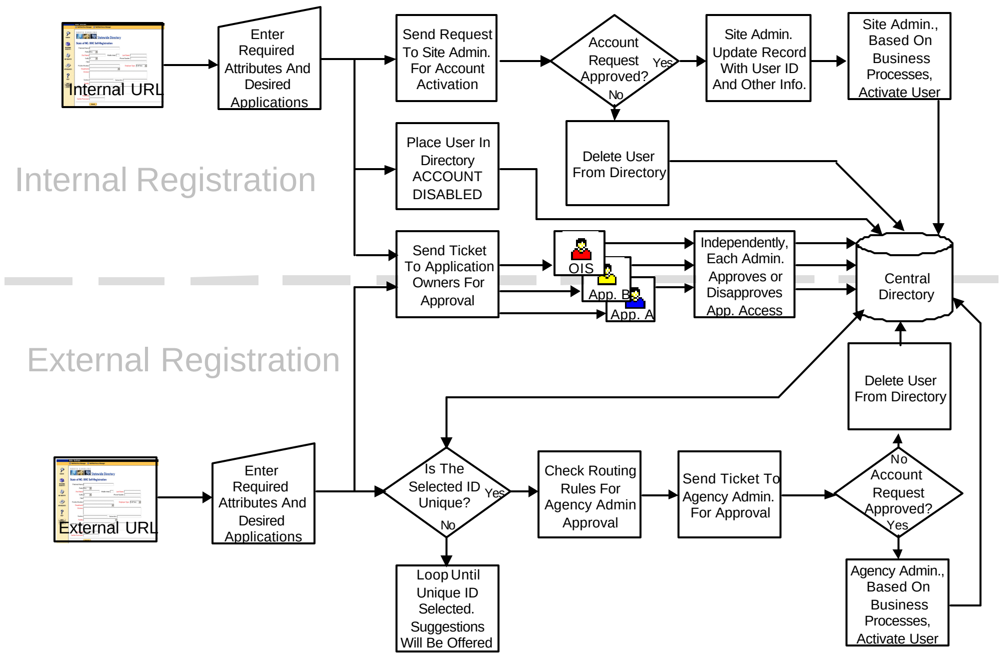

Internal URL
Enter Required Attributes And Desired Applications
Send Request To Site Admin. For Account Activation
Account Request Approved?
Yes
No
Site Admin. Update Record With User ID And Other Info.
Site Admin., Based On Business Processes, Activate User
Place User In Directory ACCOUNT DISABLED
Delete User From Directory
Internal Registration
Send Ticket To Application Owners For Approval
OIS
App. B
App. A
Independently, Each Admin. Approves or Disapproves App. Access
Central Directory
External Registration
Delete User From Directory
External URL
Enter Required Attributes And Desired Applications
Is The Selected ID Unique?
Yes
No
Check Routing Rules For Agency Admin Approval
Send Ticket To Agency Admin. For Approval
No
Account Request Approved?
Yes
Loop Until Unique ID Selected. Suggestions Will Be Offered
Agency Admin., Based On Business Processes, Activate User

Figure 14 – Example of a customized workflow

The IAMS solution provides for a robust Application Programming Interface (API) based on XML or a service loaded on a web server utilizes on HTTP header variables. Agency individual needs will dictate the appropriate approach.

IAMS is structured as a central service, providing authentication to validate user credentials (figure 15) and authorization (access control) to the resource requested by the user. More granular authorization is retained at the application and is role-based. Roles are defined, typically in the system database, which match roles defined in IAMS. IAMS will pass role information to the application through the API or HTTP header variables. The application then handles the role information to further control access to functions within the application on a more granular level.

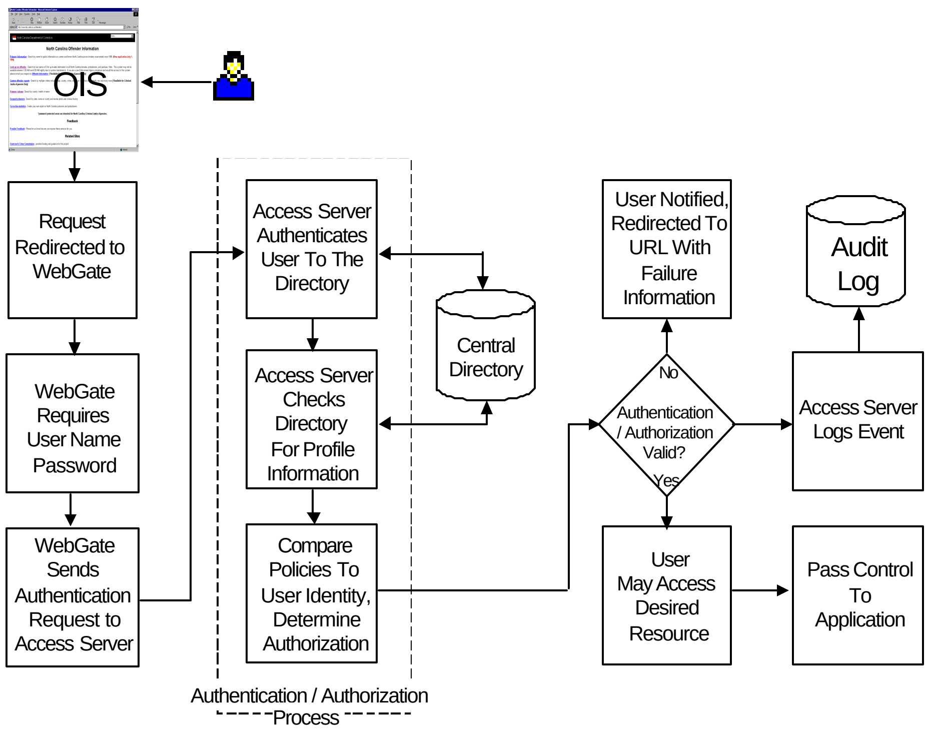

OIS
Request Redirected to WebGate
WebGate Requires User Name Password
WebGate Sends Authentication Request to Access Server
Access Server Authenticates User To The Directory
Access Server Checks Directory For Profile Information
Compare Policies To User Identity, Determine Authorization
Central Directory
Authentication / Authorization Process
User Notified, Redirected To URL With Failure Information
No
Authentication / Authorization Valid?
Yes
User May Access Desired Resource
Audit Log
Access Server Logs Event
Pass Control To Application

Figure 15 – IAMS Authentication Process

Credits

Reviewing Agencies:

1. Department of Administration
2. Office of the State Controller
3. Department of Health and Human Services
4. Department of Environment and Natural Resources
5. Office of Information Technology Services
6. Office of Enterprise Technology Services